Energy meteorology - mathematical analysis of prognoses & model development

Prof. i.R. Dr. Heinrich Werner

Online M.Sc. Wind Energy Systems
University of Kassel and Fraunhofer IWES

wes.online
ENERGY METEOROLOGY

Prof. i.R. Dr. Heinrich Werner

Arbeitsschwerpunkte

- Biology and Environment
  - Prey catching behaviour of tods
  - Species profile in flowing waters
  - Water quality monitoring / control
- industrial Production
  - Quality control print media
  - Production control laminatio
  - speech recognition
  - error detection Chip manufacturing
- Medicine
  - Spectra of effect of pharmaceuticals
  - Markers for psychic disorders in EEGs
  - Sleep profiles from EEG-Data
- economy
  - Models for stock value changes
  - Prediction of portfolio developments
- Energy management
  - Turbines, models and control
  - Weather prediction
  - Wind power prognosis
Akademischer Werdegang

31.03.1944 geboren in Utting/Ammersee
1964-1969 Studium Mathematik an der Uni Bonn
18.06.69 Diplom in Mathematik
20.02.1971 Promotion in Mathematik
1972/73 PostDoc Fellow U. of Manitoba, Winnipeg
06.02.1976 Habilitation in Mathematik
1978 Ruf als Univ. Professor(C3) an die Uni Kassel
Universelle Algebra & Kombinatorische Geometrie
1982 visiting Prof. LaTrobe Univ. Melbourne
1986 Einrichtung des Nebenfachs Informatik
Umorientierung auf das Fach Informatik: Neuronale Netzwerke
2000 Einrichtung des grundständigen, universitären Studiengangs Informatik
2009 Ruhestand
Berater des Fraunhofer Instituts IWES
Goals
1. Differentiate data file, data set and model
2. create simple visualizations
3. distinguish functional from classifying models
4. understand correctness, generalization, and verification
5. do simple data file manipulations
The aim of this lecture is to present mathematical methods for the analysis of large data files as they arise from natural or mechanical processes. A process may depend on a number of different parameters, which influence the outcome of the particular process. The data files consist of sequences of parameter values for each process step. These sequences are called the data sets, and so a data file of some process is a collection of data sets of its process steps.

To exemplify this we consider the meteorological process of weather development. The weather phenomenon appears in several different aspects, such as temperature, humidity, radiation, cloudiness, wind strength, wind direction etc., which can be considered as output parameters of the weather process. These parameters depend on a large number of further parameters such as time, position (e.g. GPS coordinates and height above sea level), surface structure (open sea, flat land, city surrounding, mountains ...) and in particular the weather conditions of the near past in the close surrounding area. These parameters are referred to as input parameters.

The weather forecast tries to give some information about the future weather development in a given area. To do this it is necessary to develop a mathematical model, which simulates the weather process. This means we need a mathematical system having the same input-output-behaviour like the weather process. The weather process itself has been observed since more than 100 years in thousands of protocol files of different accuracy and used to develop weather models by several scientists and commercial firms to produce weather forecasts. These protocol files are the data files under consideration and they contain as data sets the sequences of all relevant parameter values for a particular time and position.

These models are mathematical functions, which calculate the values of the output parameters from the values of its input parameters. Models are evaluated on the basis of the data files under consideration by calculating the errors (difference between the calculated and measured values). This means we use the models based on past experiences in order to predict the future behaviour of the modelled process. Under the assumption, that the future behaviour of the process does not differ entirely from its past behaviour and that all relevant parameters have been considered in the model, we can expect an answer of about the same accuracy the model shows on the data files of the past.

This lecture will focus on mathematical methods for data analysis and modelling relevant for the power prediction in renewable energy production.

1.1 Visualization

A very helpful tool in data analysis is graphical visualization of data files, which of course is mainly limited to 2-dimensional data or at most 3-dimensional data.
respective. Visualisation itself is not a model for the data file, however it can be used to find adequate models for the data in question.

1.1.1 Simple models

Let us start with a very simple example data file containing only 4 points in 2 dimensions

![small data file](image)

**Figure 1-1 : small data file**

We can ask for the best (constant) y-value which approximates all y-values in this point-set, which is the average of the y-values:

![constant model](image)

**Figure 1-2 : constant model**

We could also try to find the best polynomial function which approximates this point-set:
We see, that the y-distances to the model have decreased considerably by increasing the degree of the model function.

The next increase of degree changes from best approximation to exact interpolation from degree 3 on, since we started with only 4 points.

These interpolating models show a quite different behaviour on x-values between the third and fourth point, since these points are much more distant from each other than the other ones. This indicates:

Memo 1: garbage in-garbage out: We can’t expect a good model performance on parameter values, which have no examples in the data file in the neighborhood. There is no need to stick to polynomial functions as models, any other kind of mathematical function will do as well. However we could come to the conclusion, that the underlying point set is best not be interpreted by a function but as a classification:
DATA ANALYSIS AND MODELS

In this kind of model we merely answer the question, which points have low distance (high resemblance) or not.

What is the benefit of this kind of model?

The immediate information of this model is, whether a new data set belongs to one of the existing **classes** and hence is similar to one of the data sets in the underlying data file. Therefore this kind of model is not immediately useful for input-output predictions (forecasts).

However, if we enrich the classification model by function models as above for each class, we may also get a function like model for the whole data file:

The essential difference is, that the model represents a partial function, that is: not every input has a well defined output.

A not well defined output means that there may be either no output at all, like in this simple example, or more than one output for the same input in more complicated examples.
DATA ANALYSIS AND MODELS

1.1.2 Large data files

Real life examples will differ considerably from the simple example above. For example we usually have data files of much more data sets, e.g.:

![Figure 1-8: evenly spread data points](image)

This example shows a large set of data points evenly distributed over the shown space. It seems impossible to detect any dependence between the x-variables and y-variables of this particular point set. This visualization is of no help for the interpretation of data, however it indicates, that the two variables x and y are independent.

An ideal representation of a data file would be, that all points are lying exactly on a mathematical curve, such as:

![Figure 1-9: exactly functional data points](image)

This picture allows to interpret the data in the model $y=a^2x^2 + bx + c$ for suitable numbers $a,b,c$. The task remains to determine these numbers $a,b,c$.

However life is not always so simple. Firstly it may be that what we interpreted as section of a parabola might in fact be a section of a sinus curve, or even some more difficult curve, such that our parabola-model does not create the exact y-values for some of the x-values. Secondly if data are created by real life measurements we always have to struggle with the problem of measurement inaccuracy. This means that we always have to consider a small difference between the accurate value and the result of the measurement both for the variables x and y. Hence except for very special situations with very small measurement inaccuracies we can’t expect a graphical representation of measured data like above.

Much more likely are data files with visual representations similar to one of the following:
Figure 1-10: data points around a center
Here the model building task is to find the center point (marked red) and to determine a radius, such that say 90% of all data points are closer to the center than this radius.

Figure 1-11: data points around a straight line
A model here would be a suitable linear function (marked red) and a radius, such that say 90% of all data points are closer to the line than this radius.

Figure 1-12: data points around a curve
Again a model would be a suitable mathematical function (marked red) and a radius, such that say 90% of all data points are closer to the function graph than this radius.
DATA ANALYSIS AND MODELS

**Figure 1-13: non functional point set**

Even though for this point set a mathematical curve can be used as a model, this model does not represent a mathematical function (y as a function of x), since for most x values two y-values are possible in the model. Never the less it reflects a strong dependency between the two parameters.

This kind of visual representation by mathematical curves is more than one can usually expect. In real life data files lead to some kind of cloud-picture:

**Figure 1-14: data point clouds**

In such a situation a classification model might be the right approach for modeling. Another approach might be to interpret this cloud as superposition of a small number of functional models or a combination of both.

The aim of this lecture will be to provide tools for all these kinds of modeling.

1.1.3 Data of dimension three

Even though our vision is limited to 2 dimensions, we are perfectly able to navigate in 3 dimensions. There are two aids to make this orientation possible. The first is our ability to move and thereby interpret the changes in our vision as 3 dimensional phenomena. The other one is color, because objects in different distances from the viewer appear under different colors.
Figure 1-15: colored map
This phenomenon is also used in geographic maps, where different elevations are symbolized by different colors or at least different color intensities. The coloring gives a good impression of the height above or below the ground.

This mapping technique can also be used to make good visualizations of 3-dimensional data.

Figure 1-16: MATLAB surface map
In MATLAB these 3-dimensional graphics can be turned freely around so that one gets a good impression of the 3-dimensional point set. The only problem with such a surface is, that it needs very much experience with mathematical surfaces to make a good guess on the underlying formula.

Figure 1-17: MATLAB sphere
Another problem is that in a 3D graphics you see the surface of the point set but not the points behind, e.g. the above graphics of a sphere does not show the points
inside the sphere which may or may not belong to the point set illustrated by the graphics. These few examples show that 3D graphics can help to get an impression of a point set but are of only limited use to construct a model.

1.1.4 The curse of dimension

Still all our examples are much too special for the applications we have in mind. In problems about weather or power prediction or modeling a wind park the number of relevant parameters is from 20 up to several 100 rather than 2 or 3. Thus the data sets in our data files are vectors of dimension 20 or more instead of dimension 2 or 3. The large number of parameters leaves us with several difficult problems.

- Since our biological visual system is limited to 2 dimensions there is no adequate visual representation of high dimensional data.
- The classical (deterministic) methods for modeling low dimensional data files fail completely for very high dimensions since as a general rule the computational complexity of these algorithms grows exponentially with the number of variables.
- Statistical procedures base their confidence-information on the distribution properties of the a-priori data (in the data file). Since the number of necessary samples for obtaining information about special distribution properties also increases exponentially with the dimension of the problem, no proper calculation of confidence intervals is possible.
- The number of subsets of parameters, which could possibly have interdependencies among each other also increases exponentially with the dimension \(d_{\text{dim}}\) possible sets of parameters) and it is merely impossible to investigate all these interdependencies.

The impossibility of extending algorithms to higher dimensions because of the exponential explosion is called the curse of dimensionality\(^1\). To make this problem a bit clearer we consider an example.

\[^1\] Probably first quoted in “R. Bellman, “Dynamic programming”, Princeton Univ. Press (1957)” under the denotation “curse of dimensionality”.

In today’s technology the number of bits on a hard disc (say 2 TByte) is less than 1014. If we have a 20 dimensional problem and we want to have on each parameter a resolution of at least 10 possible values as supporting points, we end up with a maze of 1020 supporting points for the definition space of our problem. I.e. the file only containing the supporting points needs more than 1 Million of hard disks to host this single data file, not taking into account the time necessary to measure this immense number of data sets. (1020 seconds are about 3*1012 years compared to the 1010 years our universe exists).

It is clear, that we will not be able to do anything of the quality we are used to in two dimensions, when we go up to dimension 20 or higher. Remains the question,
what kind of quality compromise we are willing to make for a decent modeling of high dimensional data.

Looking into natural processes like the development and growth of a human body or even simpler human sensual processing we see that nature has managed to handle problems of a much higher dimensional complexity very well. We are able to recognize faces after a long time in totally different contexts and under a wide range of observation angles perfectly well. The only compromise nature makes is a lack of precision. Of course the price paid for imprecision is possible failures and errors made by the system in question. The benefit of this approach is the existence of algorithms, whose complexity does not exponentially increase with dimension, which can solve the modeling problem in an approximate way.

The aim of this lecture will be to investigate a number of such approximate algorithms subsumed under the theory of artificial neural networks.

Memo 2 approximation: Each application of an approximation algorithm calls for the investigation of possible failures and errors of the constructed model.

1.2 Models

We have mentioned the model notion several times in the examples above without having explained this notion in more detail. In this section we give some more details on models at an abstract level. The reader should bear in mind that models are one of the central themes of this lecture and only step by step he will fully understand all facets of this notion.

1.2.1 Models and data files

A Model always refers to some process and to one or more data files describing the behavior of this process. As a rule the data files contain measured values of the relevant quantities belonging to process instances. A model always is a mathematical formula or a program, such that the formula or program produces a “true”, “false” or “undefined” for any data set of the same type as the data sets in the data files of the process. We say a data set belongs to the model or the model allows for it, if the model produces “true” for this data set.

A model is correct for a data file, if all its data sets belong to the model. A data set, for which the model produces false, is called error or exception, and if the model produces undefined we speak about a failure.

In science and technology the most important question is “what happens if...”. The aim of modeling is to get answers to the following kind of questions:

- (Correctness) Does the model describe the behavior of the process?
- (Prediction) How does the process (model) run with values in a certain range?
(Outlier detection) If some of the measurements fail to belong to the model, is that because their values are exceptional (erroneous measurements or out of range)?

Etc. ...

As an example we consider bridge constructions. A load test for the maximal admissible load is best not done on the bridge itself (too expensive) but on a model. A classical 1:100 model is not suitable for load tests since length (distance values), resistance (area values) and load (mass values) have ratios 1:100, 1:10000 and 1:1000000 from model to nature. Here we need a mathematical model to perform a meaningful load test.

Note that a model does not only aim on the data file it was constructed on, but to answer the question whether a particular data set of the same dimension like in the file belongs to the model or not. Hence the model generalizes to all possible data sets of the same dimension.

1.2.2 Parameters

The quantities relevant for a process are called its parameters and so a corresponding data set is a sequence of values, one for each parameter. The parameters can be classified:

- **Accessible parameters** can be measured immediately such as time, length, temperature, etc. The (hidden) inaccessible parameters are relevant.
DATA ANALYSIS AND MODELS

quantities whose values can only be estimated or calculated by another model, e.g. internal temperature in a massive block.

Independent parameters are freely changeable by the user of the process while the dependent parameters change in consequence of the change of the other variables. The independent parameters are used as input variables in the models and the dependent variables as output.

As an example we look at a wind park, which transforms wind energy (independent input) into electrical energy (dependent output). Wind energy at the location of the wind park itself is not accessible, it has to be calculated from a complicated weather model with among other inputs wind velocity and wind direction at the various weather stations.

1.2.3 Functional models

Most models are used in the following way. We select from our process the independent variables as input for the model, and the dependent variables as output. The model itself is a program providing a sequence of functions, for each output variable one with the input variables above. For each input of the correct dimension the model produces a sequence of outputs, which is interpreted as the output, the process is going to produce in case this particular input is presented to the process.

As explained above we do not expect the model and the process to behave exactly the same, but within a given margin of error these two should coincide. In classical models there is given a mathematical proof that a model coincides with the process within a certain range. In higher dimensions, where such a proof either fails or produces useless large margins, we have to do at least some tests with our model:

- **Correctness**: For all data sets in the file the model is constructed from, the model must produce a data set within the predefined margin. A very small number of data sets in the file which have a too large distance from their model data set may be tolerated as exceptions. These exceptions should be investigated separately.

- **Generalization**: We have to produce another data file (a test file) by running and measuring the process. Again the model must produce for every data set in that file a model data set within the given margin. Very few exceptions might be tolerable but have to be investigated further.

1.2.4 Classification models

The construction of functional modes may fail either, because there is no obvious distinction between independent and dependent parameters, or the functional models we are able to construct all produce too many exceptions. In this situation it is advisable to construct a classification model for the data file.
DATA ANALYSIS AND MODELS

A classification model subdivides the members of the file into disjoint subsets (the classes) and selects from each class one member as representative. Usually different elements are appointed to the same class, only if their distance (Euclidian or any other distance measure) is very small, and the representative is chosen as close as possible to the center of its class.

The model accepts as input a data set of the dimension as in the data file and returns the representative of the class this input belongs to or produces a failure if such a class doesn’t exist. If the classes are defined via a distance measure, the model returns the representative which is closest to the input or a fail, if more than one representative is closest or the closest representatives are too far from the input.

As in functional models we test correctness (none of the data sets in the data file produces a fail) and generalization (none of the data sets in the test file produces a fail). Few exceptions may be tolerated and incorporated into new classes as new representatives of the model.

Note that a classification model just answers the question whether a given data set is close to a real process step, but it doesn’t find the appropriate output for some possible input.

1.2.5 Model construction

At this abstraction level we can’t give a detailed picture of constructing a model from a data file however we can give a hint to some fundamental techniques.

As stated above a model consists of several mathematical functions producing an output or a class representative. These functions may depend on certain constants, e.g. the coefficients in classical polynomials or linear combinations of higher functions. Therefore we call these constants the coefficients of the model.

One important technique to construct a model is to define a model as a start with arbitrary coefficients e.g. at random. This model will be far from correct or even generalizing. The aim now is to use some approximation algorithm to change the coefficients in such a way, that the model’s correctness increases a bit with each approximation step. Examples of such algorithms will be covered later in the lecture. This approximation approach is often called “adaption” ore more pretentious “learning”.

We will see, that this kind of adaption procedure does not always immediately lead to a correct model. The reasons for a failure could be:

- The adaption procedure converges too slowly and can’t produce a model in acceptable time.
- A wrong starting set of coefficient does not lead to a convergence.
- The mathematical function framework could be unsuitable for this particular data file.
- The whole problem might be algorithmically unsolvable.
DATA ANALYSIS AND MODELS

As a rule we are unable to determine, which of the reasons are responsible for a failure, there is no other way than to start many model training experiments hoping for success in at least one case.

If we have produced a correct model for some data file, there is no a priori reason, why this model should also generalize well on the test file from the same process. If this is not the case we have to go back to model construction until both correctness and generalization are satisfactory. But now both files have been used for model construction, of course to a different extent. Now, just to make sure, we use a third file from the process, the verification file, to determine the model performance on entirely new data sets the model has never experienced before. This last step is called verification and produces a measure, how reliable the model probably is.

As we learn in early programming a test can never replace a correctness proof, however in situations under the curse of dimension, where a proof exceeds all bounds of complexity, we have to settle for tests as a crutch. The price we pay is, that we can never be absolutely sure about the performance of a model and no valid error estimate is possible. This is the major drawback of adaptive model construction, but as we explained earlier we have no choice under the curse of dimension.
Goals
1. Understand the notion of artificial neural networks
2. Mathematical handling of network models
3. Basic notions topology, propagation, training
4. First neuron model
5. Learning in neural networks
NEURAL NETWORKS

2 NEURAL NETWORKS

As we pointed out above nature has developed mechanisms to handle high dimensional problems pretty well. Consider our visual system, where the eye has the retina as its sensual system. The retina consists of millions of cones and rods transforming light into nervous impulses. These impulses are led to the visual center in our brain through the nervous cells and there a picture is produced. Even though or just because the system is so high dimensional human vision is such an excellent sense and human life depends heavily on its performance.

One particular approach in high dimensional algorithms which has our human neural processing as paradigm is the theory of artificial neural networks. This theory has two diametrically opposed purposes:

![Human Brain](image)

**Figure 2-1: the human brain**

- One is to construct models of real life processes in nervous systems. We aim on understanding the function of brain and nerves e.g. to be able to build intelligent prostheses for defect parts of the human body.
- The second is the construction of computer systems modeling the nervous data processing. Here usability is the main guideline rather than exact similarity.

Although the first motive is far more interesting than the second, we stick to the second purpose, since it fits exactly into our considerations of adaptive algorithms.

2.1 The biological antetype

![Neuron Mesh](image)

**Figure 2-2: neuron mesh**

We have a look at the function of nervous systems with the brain as most prominent system but also the optic nerve and others. All these form a dense mesh of connected processing cells which are called neurons.
Figure 2-3: neuron interaction
The neurons send their information as impulses over the connections to a large number of other neurons and so on and they all work in parallel. The speed of a single neuron is not particularly high – about a million times slower than a modern processor – nevertheless neural systems work highly efficient because of parallel processing and extremely high connectivity.

Figure 2-4: neuron
A neuron is a cell with a compact centre (soma) some branching connections (axon) for signals to other neurons which and many also branching connections (dendrites) for signals from other neurons.

Figure 2-5: synapse
The interconnections between the axons and dendrites are the synapses which are mainly responsible for the correct information transfer. Unlike in a computer the transfer through the synapses changes the information somewhat. Synapses can be excitatory or inhibitory which means that they reinforce or diminish the strength of reaction in the destination neuron.
NEURAL NETWORKS

Figure 2-6: neuron connection
A schematic view on a typical neuron shows the following behavior:

- A neuron has multiple inputs (dendrites) leading information to its center cell (soma).
- The center takes a degree of activation depending on the input signals.
- If the activation exceeds a certain threshold (also called bias) it is transported through the output line (axon) to many connections (synapses).
- The connections change the output information according to their strength (also called weight) into the input of the connected neurons.


2.2 Numerical neuron models

Figure 2-7: neuron scheme
A schematic view on the neuron shows an object receiving several inputs through weighted connections (weights $x_1, x_2, x_3, \ldots, x_n$ and $-b$), where $b$ is the bias of the
In all numerical neuron models the inputs \((i_1, i_2, i_3, \ldots, i_n, \text{ and } 1)\) are multiplied with the corresponding weights (resulting in \(x_1^*i_1, x_2^*i_2, x_3^*i_3, \ldots, x_n^*i_n, \text{ and } -b\)) and almost all neuron models use the simple summation to calculate the activity relative to the bias. Very few examples also use a different activation function \(\Sigma\), but in this lecture as a standard we use the simple summation for \(\Sigma\).

The calculation leads to \(x_1^*i_1 + x_2^*i_2 + x_3^*i_3 + \ldots + x_n^*i_n - b\) the scalar product of the input vector and the weight vector minus the bias or shortly the scalar product of the extended input- and weight-vectors.

\[
x_1^*i_1 + x_2^*i_2 + x_3^*i_3 + \ldots + x_n^*i_n - b
\]

\[
(x_1, x_2, x_3, \ldots, x_n)((i_1, i_2, i_3, \ldots, i_n) - b)
\]

\[
(I, x_1, x_2, x_3, \ldots, x_n)((-b, i_1, i_2, i_3, \ldots, i_n))
\]

**Formula 1 : activation**

The neuron activity relative to the bias is then transferred into the output of the neuron by a function \(\zeta\), which exists in very different forms as we see later on. This function \(\zeta\) is called transfer function or output function of the neuron.

\[
\zeta(\text{activation})
\]

\[
\zeta(\text{weight * input - bias})
\]

\[
x_1^*i_1 + x_2^*i_2 + x_3^*i_3 + \ldots + x_n^*i_n - b)
\]

**Formula 2 : output**

The above graphical representation is even more simplified in MATLAB, where each sequence or matrix is reduced to a single object:

**Figure 2-8 : MATLAB neuron scheme**

We see one input sequence led to the weight matrix \(W\) added to the bias \(b\) and then transformed by some transfer function symbolized by a graphic icon. The dimensions of the different arrows and matrices will become clear from the complete network diagrams later.

**2.3 Numerical net models**
Having established numerical models for a single neuron, we now give a numerical interpretation of a complete neural network. We will give a description of the connection structure of the network, which is also called its topology. Since we are interested in a good numerical algorithm rather than an exact model of our brain, our net models will be of much lower complexity than the net structure in a human brain.

![Neuron net example](image)

**Figure 2-9: neuron net example**

2.3.1 Topology

A neural net consists of a number of neurons and several connections. There are two special neuron types, the input neurons, whose only connections into the unit come from outside, and output neurons, whose only connections from the unit go to the outside. All the other neurons are called hidden units or internal units. The arrangement of neurons and connections is called the **topology** of the network.

The above example shows a strictly layered neural network with one input layer with 3 neurons, one hidden layer with 6 neurons and one output layer with 2 neurons in classic graphical representation. MATLAB has a much simpler representation for this:

![Neuron net example MATLAB](image)

**Figure 2-10: neuron net example MATLAB**

2.3.2 Propagation
The term propagation describes how the input is transferred into the network and how the values of neuron activations and outputs are calculated, in particular how the output values of the whole network are determined. The most important information is, how each single neuron produces its activation from its inputs and its bias (the activation function $\Sigma$) and how the output is calculated from the activation (the output-function or transfer function $\zeta$). Via the topology we can calculate the activity and the output neuron by neuron until for all output neurons their output is calculated. The propagation works in a unique way, if the network has no feedback, i.e. we can number the neurons in such a way, that all connections go from a neuron of lower index to a neuron of higher index. If feedbacks occur, we need a propagation rule about the sequence in which the neuron activations should be calculated and about a stopping condition, when the calculation can or must be ended.

There might be another complication for the propagation by some pre-processing of the input, before it is fed into the network, and/or some post-processing of the output before it leaves the network. Pre-processing might be selection of only some of the available channels or scaling all input channels to values between 0 and 1. Post-processing might be e.g. rescaling output values.

2.3.3 Adaption

Finally each neural network type needs an adaption algorithm to change the networks behaviour into the desired direction. For this we need a data file and information, which behaviour the network should show on this data file, if the actual network’s behaviour is not good enough, the adaption algorithm should change the network properties in such a way, that its behaviour improves. We discuss this issue in the chapter 2.6. under the name of learning or training.

2.4 The feedback problem

The example in Figure 2-9 and 2-10 has a very special structure, where all information is transferred from one layer to the other beginning from input leading to the output only in one direction. For each input each neuron is only activated once to produce the final network output. This means, that we have no feedback within this network in contrast to biological neural networks, which usually contain a large number of feedback connections.
Figure 2-11: neuron net with feedback

In the theory of artificial neural networks there are also many models with feedback, but only few of them will play a role at the end of this lecture.

As we know from basic automata theory any kind of memory construction needs feedback; without feedback we can only represent pure mathematical functions. Figure 2-11 contains two “memory cells” containing the momentary output. Considering networks with feedback is much closer to biological neural networks, however the numerical problems increase dramatically with the size of the networks.

The first obvious problem comes from the fact that the output is not calculated by running through each neuron of the net only once but over and over again. In the above example the memory cells and the output cells have to be recalculated repeatedly. Since the feedback changes the activities of the neurons a final output can only be reached, if the feedback doesn’t change the neuron activities any more. For a net with feedback an input starts a possibly infinite series of activity changes, which in ideal situations can end up in a final stable state, where no more changes of activity take place. The normal situation however is an infinite aperiodic or periodic series of changes.

The second problem is that in networks with feedback it is much more difficult to reach a well defined output and hence the adaptation of the weights to obtain a correct model is far more complicated than without any feedback.

For this reason in this lecture we abandon feedback networks and only use feed forward networks as our models. Nevertheless in the final sections we will also construct models with some restricted feedback, which only results in memory activities rather than repeated recalculation of neuron activities. With this kind of approach we will be able to circumvent the numerical instability of arbitrary feedback.
2.5 McCulloch-Pitts neuron

Figure 2-12: McCulloch-Pitts Neuron

Historically the first approach to a neuron model was the paper (W. Pitts, 1943) constructing a neuron model for the proof that a human brain can calculate Turing\(^2\) computable functions. The model was based on 0-1-input and output such that if it fired 1 if the sum of Inputs exceeded the bias \(\theta\). The second aspect was the distinction of two types of connections. The \textit{excitatory} connections propagate the output of its start neuron unchanged into the destination and thereby increase the tendency of the destination neuron to fire. The \textit{inhibitory} connections, which propagate the negative output of its start into the destination and thereby decrease the tendency of the destination neuron to fire (denoted by the Symbol \(-\)).

The main goal at this time was to prove, that all notions of computability developed by various authors in fact coincided. The networks constructed by McCulloch and Pitts were the standard logical functions, which were known to combine to each Turing computable function. In this approach the networks for the functions in question had to be programmed like the Turing machines.

Figure 2-13: McCulloch-Pitts XOR Net\(^3\)

2.6 Learning, Training

It took more than 10 years before the investigation of artificial neural networks made an essential advancement by replacing the strict programming by flexible

\(^2\) The Turing machine was defined by Alan Turing in 1936 as a universal calculator in order to define computability. The same Alan Turing during the second world war had an important part in cracking the code of the German ENIGMA, which made the German U-boat fleet noneffective.

\(^3\) If \(x, y\) are different (0,1 or 1,0) one of the first two neurons fires 1 and hence the output neuron fires 1. If \(x, y\) are equal (0,0 or 1,1) then all three neurons don’t fire and hence the output is 0.
adaption algorithms, which in view of the biological paradigm were called learning algorithms. This new approach did not have exact calculation in mind but approximation instead.

Unlike the McCulloch Pitts networks, where a 1-step programming leads to the network with the desired properties, nowadays an artificial neural network is initialized without any view on its input-output-properties. Then an optimization procedure is used to step by step improve the network properties. With a bit of exaggeration this optimization is called training or learning.

The behavior of a neural network is determined by 3 aspects:

- The functions each neuron uses to calculate its activation and output.
- The network topology, i.e. the number of neurons and the way they are connected.
- The weights used for the calculation of the activations.

In general the neuron functions and the net topology are a matter of layout prior to the training process; hence the actual training is a process adapting the connection weights and biases to the underlying data file.

In the theory of artificial neural networks there also exist some concepts of training for the network topology or even the neuron functions, however because of their computational complexity we will not pick them as a theme in this lecture.

**Memo 3 Training**: For this lecture training always means changing the weights of a network in an appropriate manner.

### 2.6.1 Supervised vs. self organized

There are two opposite methods of training.

One is the **supervised** method, which needs a teacher who knows for each input a correct output. An input is fed into the network and it produces the net-output. The teacher compares the net output with the correct output and calculates the error made by the net. A training step is now performed with the aim to reduce the error; that is the adapted net after the training step will produce an output on the same input with smaller error. These training steps are performed for all inputs from a given training file and so passing through the training file several times until the total error over all inputs is small enough. This type of learning we also call **output-oriented** or **error driven**.

The second method is **self organized** training, where we are not interested in the correctness of some output, but in the similarity or dissimilarity of inputs. The training algorithm aims at a classification of the inputs according to some distance measure rather than the minimization of some output error. As a consequence it is not clear at which point such an algorithm has to end, because there is no problem independent reason, why one classification should be better than another. Since output does not play any role in this kind of learning, it is also called **input oriented** or **distance driven**.

On the base of a single connection between McCulloch Pitts neurons but with arbitrary numerical weights we can exemplify these two types of training by the “delta rule” and the “Hebb rule” as follows:
In the simple case of McCulloch Pitts neurons the only possible information along this connection is 0 or 1 multiplied by the weight $w$. If the information from the start neuron is 0 its impact on the destination neuron is 0, no matter what value $w$ the weight has, hence in this case a change of the weight has no effect on the net behaviour.

**The delta rule (supervised)**

Each connection in the network is trained by the same rule:

**Memo 4 Delta**: If the output of the connection start neuron is not 0 and the actual output of the destination neuron is not the target output, the weight $w$ of the connection is decreased (or increased respectively) in case the actual output is higher (or lower) than the target output: $w \pm \Delta w$.

It is the matter of the concrete training algorithm to determine the exact amount $\Delta w$ by which the weight $w$ has to be changed. In any case this rule makes, that with the same input the network will reduce its error somewhat.
NEURAL NETWORKS

The Hebb rule (self organized)
The Psychologist Donald Hebb (Hebb, 2002) has interpreted in 1949 the training effects of a human brain:

**Memo 5 Hebb**: If the start neuron of the connection has output not 0 and the actual output of the destination neuron equals to the start neuron output, the weight \( w \) of the connection is increased \( w + \Delta w \).

This means that the connection is strengthened in case both neurons fire, in short this rule can be formulated: *what fires together, wires together*.

These two rules form the basis for virtually all training algorithms used in the theory of artificial neural networks. While the Hebb Rule is motivated from the behaviour of the human brain, the Delta Rule is motivated by effective computation and fast approximation.

2.6.2 Online vs. offline

Learning algorithms are stepwise applied to the network weights in the following manner using the training file:
The next input is read from the data file – if the reading position is at the end of file the file is started from the beginning again and fed into the network and the weight changes \( \Delta w \) are calculated for each connection of the network.
The normal method is to immediately replace all \( \Delta w \) by \( \Delta w + w \) and then proceed to the next input from the data file. This method is called **Online Learning** or **Learning by Pattern**.

The extreme opposite of this method is **Offline Learning** or **Batch Learning**, where all the \( \Delta w \) values for each input are added up and the replacement of all \( \Delta w \) by \( \Delta w + w \) is postponed to the end of file.

A compromise between these two extremes is **Block Learning** or **Learning by Epoch**, where the training file is subdivided into smaller parts (the blocks or epochs) and the replacement \( \Delta w \) by \( \Delta w + w \) is always done at the end of an epoch.

The different methods produce slightly different results and are of different numerical quality, however there is no problem independent reason to prefer one to the other.

All Methods repeatedly go through the training file, hence the algorithm has to have an own mechanism to stop the training, e.g. if the number of steps exceeds a given bound or the error (or some other algorithm relevant value) falls below a certain margin.

2.6.3 Correctness and Generalization

Having trained a network the question arises, whether it has learned successfully. The easiest way to answer this question is to have some error measure on the network. In the approximation case this usually is the distance between actual output and the target output. In classification nets one could use the distance of an input to a representative member of its class.
The minimum requirement for successful learning would be, that for all members of the training file the error measure is below a given error bound. This is called correctness of the learning procedure. Usually the stopping condition for a training algorithm is just this condition. Unfortunately this is not sufficient, to call a learning process successful.

As an example we look at a mathematics class at school. The teacher has given a block of say 10 addition tasks to his pupils. If somebody mastered all tasks, the teacher hopes that student has understood addition, but it may well be, that he has only learned by heart and didn’t quite understand, what he was doing. So the teacher will have more tasks to hand out for control. Only if his scholars are able also to master these tasks as well, he will be convinced that they really have learned addition. However still this is only a piece of evidence but not a proof for the pupil’s abstraction and generalization ability in addition tasks.

As in our example we expect a successfully trained network not only to handle the training file correctly, but also all other data sets, which can be obtained from the same process as the training file. This desired ability of the network is called generalization and it can be tested similarly to the above example.

![Generalization Test Diagram](image)

**Figure 2-17 : Generalization Test**

We need a second data file used for testing. We feed the inputs from the test file into the net and calculate the error measure for each input. If all these values are below the error bound we accept this network as sufficiently generalizing. The point is, that the test file consists of inputs, the network hasn’t been trained on, and still the network must be able to act correctly also on these inputs. If the test succeeds we have no proof for generalization but a piece of evidence. If the test fails, we have to do training once more, maybe even with an amplified training file, until the generalization test succeeds. Even if we haven’t used data from the test file for the further training, still we have used that file indirectly for evaluating the training process. In this case we need even a third data file to finally verify the training success. This verification file is then used the same way as the test file before to finally judge the performance (error level) of the trained net.

### 2.6.4 Black Box

Especially in the situation, where a network models some functional input-output-behavior, we need correct input-output pairs for training, test, and verification. The way we want to use this model however is, that we generate the output from a
new input, for which we don’t know the correct output in advance. Our networks will give an answer to any input we choose to feed into the network, but unfortunately there is no obvious way to analyze, how the network came to its answer and how sure I can be about its correctness. This seems to be a system immanent deficit, as e.g. in face recognition a human brain is not analyzable how it came to its decision of similarity or dissimilarity. We have to live with this black box property of neural networks still some scientific effort goes into analyzing the confidence in the performance of neural networks.
Goals
1. Definition of the perceptron
2. Training and convergence
3. Mathematical background
4. The XOR problem
5. Matlab realization.
3 PERCEPTRON

As the historically first and most simple version of an artificial neural network is the perceptron invented by Rosenblatt (Rosenblatt, 1958), which is a neural net with only input and output neurons.

3.1 topology

If we consider some output neuron, we see, that the output value is not influenced by any of the weights of connections leading to any of the other output neurons. This means, for a mathematical analysis of perceptrons, we can restrict ourselves to those having only a single output neuron.

3.2 propagation

The original paper also used the simplest propagation rule, namely the activation is defined as the sum $x_1*w_1 + x_2*w_2 + x_3*w_3 + \ldots + x_n*w_n - b$ of the weighted inputs minus the bias (scalar product rule) which can be calculated as the scalar product $x_{ext}^*w_{ext}$ of the extended input $x_{ext}^* = [x_1 \ x_2 \ x_3 \ \ldots \ x_n \ 1]$ and the weight extended by the negative bias $w_{ext}^* = [w_1 \ w_2 \ w_3 \ \ldots \ w_n \ -b]$.

The output is calculated $\zeta(activation)$ with the 0-1-step function $\zeta(a) = (a \leq 0) ? 0 : 1$ (McCulloch Pitts).
### 3.2.1 Calculation

Note that propagation expects a weight/bias row vector and a matrix of input rows and produces the activity and the output of the perceptron.

\[
\text{table} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}
\]

contains all possible 0-1 inputs in 2 variables and so the extended Matrix is

\[
\text{table}_{\text{ext}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}
\]

with the weight/bias row \( w = [2 \ 1 -1] \) it produces

- the activities \( a = w \cdot \text{table}_{\text{ext}}' = \begin{bmatrix} -1 & 0 & 1 & 2 \end{bmatrix} \) and hence
- the outputs \( y = \zeta(a) = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \)

which is the output of the second projection \( \pi_2(x,y) = y \).

This means that the perceptron with weights 2, 1 and bias 1 (note that the weight/bias row contains the negative bias \(-b\)) is a model for the projection \( \pi_2 \).

### 3.3 Training

The perceptron is trained in a supervised manner based on the delta rule.

\[
\text{Figure 3-4 : Perceptron test}
\]

We assume we have a data file with its data sets consisting of the training inputs together with the associated target (desired output) as a last entry. The network is initialized by weights \( [w -b] \) zero (random weights or weights from a prior network version are possible as well).

Going several times through the whole training file in each training step we read the next data set \([x \ t]\) (input target) from the file, feed the extended input part...
PERCEPTRON

[x 1] into the net input, propagate the input through the net producing an activity \( a = [x 1]^* [w - b] \) and a net output \( y = \zeta(a) \), and compare \( y \) to the target output \( t \) from the actual data set defining \( \text{error} = \text{output} - \text{target} = y - t \).

There are 3 cases possible:

1. \( y = t \) \( \Rightarrow \text{error} = 0 \); both values coincide.
   Then no training is necessary: \( \Delta w = 0 \), \( \Delta b = 0 \).

2. \( y = 0 \), \( t = 1 \) \( \Rightarrow \text{error} = -1 \); activity and net output too low.
   The weights \( w \) have to be increased by some \( \Delta w \) and the bias decreased by \( \Delta b < 0 \) to increase activity.

3. \( y = 1 \), \( t = 0 \) \( \Rightarrow \text{error} = +1 \); activity and net output too high.
   The weights have to be decreased by some \( \Delta w \) and the bias increased by \( \Delta b > 0 \) to decrease activity.

We use the delta rule as training paradigm, that is defining \( \text{error} = \text{output} - \text{target} \) and a learning rate (step width) \( \eta > 0 \) for each weight \( w_i \) we choose \( \Delta w_i = \eta \text{error} \) and \( \Delta b = -\eta \text{error} \).

However as we pointed out in chapter 2 in case \( x_i = 0 \) a change of weight \( w_i \) is useless, so in this case we have \( \Delta w_i = 0 \).

Since \( x_i = 0 \) or \( x_i = 1 \) we can combine these two cases by \( \Delta w_i = \eta \text{error} x_i \) and the calculation \((w_i - \Delta w_i) x_i = w_i x_i - \eta \text{error} x_i^2 \), \( -b - \Delta b = -(b + \eta \text{error}) \) shows that after the change activity increases in case 2 and decreases in case 3 as desired.

We can treat all indices together in vector notation
\[ \Delta w = \eta \text{error} x, \quad w_{\text{new}} = w_{\text{old}} - \Delta w, \]
where \( w \) and \( x \) are the extended vectors \([w_1 \ldots w_n - b]\) and \([x_1 \ldots x_n 1]\).

This formula with the special choice \( \eta = 1 \) is called the \textit{perceptron learning rule}: \( \Delta w = \text{error} x, \quad w_{\text{new}} = w_{\text{old}} - \Delta w \).

We say that the training algorithm \textit{stops}, if the perceptron learning rule produces \( \text{error} = 0 \) for all data sets in the training file.

Note that we obtain the corresponding formula with a + instead of the -
\[ \Delta w = \eta \text{error} x, \quad w_{\text{new}} = w_{\text{old}} + \Delta w \]
in case we define the error the other way \( \text{error} = \text{target} - \text{output} \).

3.4 The perceptron convergence theorem

\textit{Given a data file} \( T \) \textit{and a corresponding perceptron} \( P \).

\textit{The perception training algorithm on} \( P \) \textit{with file} \( T \) \textit{stops after a}
Incautious readers have misinterpreted this theorem as “a perceptron can learn anything”, which is as wrong as can be. The important part of this theorem is the proviso, that T in principle can be modeled by a perceptron, only under this assumption the perceptron training algorithm will succeed in a finite number of steps.

The mathematical phrase “in a finite number of steps” also needs some consideration. In praxis the finite number of steps is of no help, since it may be too large to be done in the lifetime of the calculator. What we would need is a (problem dependent) upper bound to the number of training steps, but unfortunately our proof below does not provide such an upper bound.

We can reformulate the perceptron convergence theorem in the following way:

Given a data file $T$ and a corresponding perceptron $P$.

If the perception training algorithm on $P$ with file $T$ does not stop after a number $N$ of steps one of the two alternatives is true:

1. $T$ has no perceptron model at all.
2. The training process needs more than $N$ steps.

Unfortunately there is no obvious way to distinguish between the two alternatives. Before we exemplify this theorem we first give a proof of the theorem. The proof will help understand this theorem but it is not necessary to memorize this proof.

Given a data file $F$ and a corresponding perceptron $P$.

In a first step we read the lines of $F$ into rows of a matrix $T$. Then we leave the rows $[x_1 \ x_2 \ x_3 \ ... \ x_n \ 1]$ unchanged but we change the rows $[x_1 \ x_2 \ x_3 \ \ldots \ x_n \ 0]$ of $T$, which contain a 0 in the last position to their negative with a $-1$ in the last position $[-x_1 \ -x_2 \ -x_3 \ \ldots \ -x_n \ -1]$

The proviso above assumes that there exists some weight/bias vector $w$ such that propagation of all rows in $T$ produces errors 0. This means the vector $T^*w'$ has all entries > 0, because the rows in $T$ containing a $-1$ as the last entry are precisely the negative of an extended line of $F$ with target 0 (activity < 0).

Now the perceptron training works as follows:

The weight vector $u$ is initiated by $u_0 = \text{zeros}(1,n+1)$

Circulating through the rows $t_i$ of $T$ each row is propagated into the perceptron: $\text{act} = t_i^*u_k'$ and if $\text{act}<0$ the learning step is $u_{k+1} = t_i^*u_k$ otherwise no learning is necessary.

The learning ends, if $T^*u_k'$ has all entries $\geq 0$.

In order to show that training will end we define some special numbers: $t_i^2 = t_i^*t_i'$,
PERCEPTRON

\[ m = \min(t_i \ast w' \mid t_i \text{ row of } T) > 0 \text{ , and} \]
\[ M = \max(t_i^2 \mid t_i \text{ row of } T). \]

By construction \( u_k = t_{i_1} + t_{i_2} + \ldots + t_{i_k} \) is a sum of \( k \) rows of \( T \) and for all \( k \) \( \text{act} = t_{i_k} \ast u_{k-1} < 0 \). Thus we get

(formula 1) \[ u_k \ast w' \geq k \ast m. \]

Since \( (u_k \ast w')^2 \leq u_k^2 \ast w^2 \) we have

(formula 2) \[ u_k^2 \ast w^2 \geq k^2 \ast m^2 \text{ and hence } u_k^2 \geq k^2 \ast m^2 / w^2. \]

Since \( u_{i_{k+1}} = t_i + u_k \) we get for all \( k \):

\[ u_{i_{k+1}}^2 = (t_i + u_k)^2 = t_i^2 + u_k^2 + 2 \ast (t_i \ast u_k') < t_i^2 + u_k^2 \text{ and hence} \]

(formula 3) \[ u_k^2 < t_i^2 + t_{i_2}^2 + \ldots + t_{i_k}^2 \leq k \ast M. \]

Combining these formulae we get

\[ k \ast M > u_k^2 \geq k^2 \ast m^2 / w^2 \text{ which implies } k < M \ast w^2 / m^2, \text{ so the number } k \text{ of training steps is bounded above. Unfortunately without knowing } w \text{ we don't know } m \text{ either and have no chance to calculate this upper bound of } k \text{ from this proof.} \]

3.5 programs

As a demonstration we present here MATLAB programs realizing the propagation of data in a perceptron and the perceptron training.

3.5.1 Simple propagation

We start with a simple version which processes only a single input (row vector) and produces as output a single value.

```matlab
function [ out ] = P_propagate( wb, inp )

% P_propagate simulates a perceptron with [weights -bias] = wb
% P_propagate expects a weight vector wb containing the connection weights
% and as last item the negative bias.
% The row vector inp has length(wb)-1.
% P_propagate returns the outputs of the perceptron

weightno = length(wb)-1; % number of weights (without bias)
T = [inp 1]; % extended input for the bias is set to 1
activity = T \* wb'; % the scalar product activation rule
out = 0; % initiate the value out
if (activity>0) out = 1; % calculate the output by the
end; end;
```

Program 3.5-1: Perceptron_propagate

3.5.2 error calculation
A necessary extension of this is the calculation of error, where we assume that each input is extended by the corresponding target. Furthermore we want to allow a whole sequence of input rows instead only a single one.

```matlab
function [ err ] = P_error( wb, inp )
% P_error simulates a perceptron with [weights -bias] = wb
% P_error expects a weight vector wb containing the connection weights and as last item the negative bias.
% The matrix inp has the same number of columns as wb and its last column is treated as the target output too many columns are ignored.
% P_error returns the error vector of the output neuron.
% Note that the error is simply the output in case all targets are set to 0 hence P_error can also be used for propagation of input sequences.
% Since all errors are 1 or -1 the square: error'*error is the number of error(i)#0 and can be used to count the number of errors.
wbno = length(wb);  % number of weights and bias
rowno =size(inp,1);  % number of simultaneous inputs{rows of inp}
colno=size(inp,2);  % length of each row of inp
if (wbno>colno)
    T = [inp zeros(rowno, wbno-colno)];  % too few input columns are completed
else
    T = inp(:,1:wbno);  % too many input columns are ignored
end;
% T has rowno rows and wbno columns
target = T(:,end)';  % the target output (or zeros) saved in target
T(:,end) = ones(rowno,1);  % extended input for the bias is set to 1
activity = T*wb';  % the scalar product activation rule
output = zeros(rowno,1);  % same dimension as activity
for (i=1:rowno)  % calculate the output
    if (activity(i)>0) output(i) = 1;
    else output(i) = 0;  % McCP_step function(activity)
end;
err = output - target;  % if target = zeros then error is the output
end;
```

**Program 3.5-2: Perceptron_error**

Note that the same function can be used to propagate a whole sequence of inputs provided the target column of `inp` is constant zero or completely absent, in this case `err` contains the perceptron outputs as column vector.

### 3.5.3 online learning

The perceptron learning algorithm is at first defined as an online algorithm treating input by input.

```matlab
function [ wb ] = P_online( train)
% p_online takes a matrix with training data and trains a perceptron, and returns the weight bias vector constructed by the training.
```
PERCEPTRON

```matlab
% p_online accepts a matrix train with the first columns treated as input
% and the last column as target output.
P=0;                % pointer to the last used row of train
L=size(train,1);    % number of rows of train
N=size(train,2);    % length of rows of train (number of columns)
maxstep=100;        % max number of learning steps
wb = zeros(1,N);    % initialize weights and bias by zeros
D=0;                % number of consecutive weights with error = 0
step=0;             % number of learning steps
while (D<L)&&(step<maxstep)
    if(p==L) p=1; else p=p+1; end        % next p, after L follows 1
    TL=train(P,:); TL(N)=1;                % input 1 for bias
    error = P_error(wb,train(P,:));       % calculate error
    if (error == 0)   D=D+1;
    else   wb=wb-error*TL;
    end
    D=0;                                  % create L consecutive errors=0
    step = step+1;                        % next learning step
end
end
```

Program 3.5-3: Perceptron_online_learning

The number maxstep is necessary for the case that learning is not successful since otherwise the program would run into an infinite cycle.

Note that the training data completely determine the network topology:
1 output neuron, no of columns many input neurons.

As an example we try the perceptron online training on the logical AND:

```
andtable = [0 0 0; 0 1 0; 1 0 0; 1 1 1];  % table of the logical AND
```

P_online(andtable) produces the output 2 1 -2 for the two weights 2 1 and bias 2.

Protocolling the whole training process we get the following table:

<table>
<thead>
<tr>
<th>weights</th>
<th>-bias</th>
<th>inputs</th>
<th>ext</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0</td>
<td>1 1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
| 0 0 0   | 0 0   | 1 1    | 1   | -1    | step 01
| 0 0 0   | 0 0   | 1 1    | 1   | -1    | step 01
| 1 1 0   | 0 0   | 1 1    | 1   | 1     | step 02
| 1 1 0   | 0 0   | 1 1    | 1   | 1     | step 03
| 1 0 -1  | 1 0   | 1 1    | 1   | 0     |
| 1 0 -1  | 1 0   | 1 1    | 1   | -1    | step 04
```
Indeed, $P_{\text{error}}([2 1 -2], \text{andtable(, 1 2)}) = [0 0 0 1]$ as desired. This table can be understood as follows:
The extended input is propagated and produces an error 1, -1, or 0. depending on the error in the last column the extended input is subtracted from, added to the weight_bias vector or the weight_bias vector remains unchanged.
This procedure is repeated until either the number of changes exceeds maxstep or each possible training input produces error 0.

3.5.4 offline learning

It is also possible to realize the perceptron learning as an offline training: in this case the whole training matrix is propagated into the perceptron obtaining an error vector (as column vector). Since the items in error are 1, 0, or -1 the expression $\text{error'}*\text{error}$ counts the number of inputs producing a wrong output. Training ends, when $\text{error'}*\text{error}=0$ is achieved.
Again in case training is not successful we need a number maxstep to end training anyway after at most maxstep training steps.
function [ wb ] = P_offline( train )
% P_offline takes a matrix with training data, trains a perceptron, and
% returns the last weight vector constructed by the training.
% P_offline treats the matrix train with the first columns as input
% and the last column as target output. The training is offline.
L=size(train,1);  % number of rows of train
N=size(train,2);  % length of rows of train
maxstep=100;  % max number of learning steps
wb = zeros(1,N);  % number of actual learning steps
step = 0;
T = train;
T(:,end) = ones(L,1);  % extended inputs: replace target by 1
D = L;  % Number of errors initialized by L>0
while ((D>0)&&(step<maxstep))
    error = P_error(wb,train);  % error calculation with targets
    D = error'*error;
    if (D>0) wb = wb-error*T;  % weight calculation with extended inputs
    step = step+1; end;
end;
end

Program 3.5-4: Perceptron_offline_learning

As an example we try the perceptron offline training on the logical AND:

andtable = [0 0 0; 0 1 0; 1 0 0; 1 1 1]  % table of the logical AND

P_offline(andtable) produces the output 1 1 -1 for the two weights 1 1 and bias 1.
protocolling the whole training process we get the following table:

<table>
<thead>
<tr>
<th>weight</th>
<th>-bias</th>
<th>error’</th>
<th>(-T*error)’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0 -1</td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1 0</td>
<td>-1 -1 -3</td>
<td></td>
</tr>
<tr>
<td>0 0 -2</td>
<td>0 0 0 -1</td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 -1</td>
<td>0 0 0 0</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>final output</td>
<td>errors 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indeed, P_error([1 1 -1], andtable(:, 1 2)) = [0 0 1] as desired.
3.6 training failures

The convergence theorem and the above examples support the misleading opinion, that a perceptron could learn any problem whatsoever. The next example shows that this assessment is wrong. An elementary function like the logical inequality (exclusive or) cannot be learned by a perceptron. In fact a geometrical analysis has shown that a perceptron is only capable of separating a point set by a linear subspace into two parts (output 0 or 1). This insight has stopped any research and development in neural network theory for about 10 years, until entirely new topologies and learning algorithms had been developed.

As an example we consider the logical XOR function \( X\text{OR}(x,y) = 1 \) if \( x \neq y \)

### 3.6.1 The XOR table

The XOR table

\[
xortable = \begin{bmatrix}
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{bmatrix}
\]

\[
P_{\text{offline}}(xortable) \text{ does not end before maxstep training steps:}
\]

<table>
<thead>
<tr>
<th>weight</th>
<th>bias</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 -1 -1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 2</td>
<td>1 0 0 1</td>
<td>% periodic</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 -1 -1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 2</td>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 -1 -1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 2</td>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>etc., periodic repetition of the first two lines</td>
<td></td>
</tr>
</tbody>
</table>

\[
P_{\text{offline}}(xortable) \text{ does not end before maxstep training steps:}
\]

<table>
<thead>
<tr>
<th>weights</th>
<th>bias</th>
<th>inputs</th>
<th>ext</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>+ 0 1 1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>+ 0 1 1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>+ 1 0 1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>- 1 1 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 0 1</td>
<td>1</td>
<td>% periodic behaviour</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>+ 0 1 1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>+ 1 0 1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This example shows, that the perceptron learning algorithm is not successful for the XOR problem, hence by the convergence theorem there is no solution to the problem at all.

### 3.6.2 Visualization by lines

Note that the equation \( w_1 x + w_2 y - \text{bias} = 0 \) is equivalent to \( w_1 x + w_2 y = \text{bias} \), which describes a straight line in the x-y-plane. This straight line is the border line, where the output of the corresponding perceptron switches from 0 to 1 or vice versa:

![Figure 3-5: border line](image)

It is quite obvious that such a configuration never can solve the XOR problem. No matter how we position a straight line, we cannot get both diagonal corner pairs on different sides of the line.

![Figure 3-6: XOR function](image)

The only way to solve this problem would be to use 2 different lines:
This approach needs 3 cooperating perceptrons arranged into 3 layers:

The first two perceptrons can be equipped with the weight/-bias vectors \([2 \ 2 -1]\) and \([-2 \ -2 \ 3]\) which represent the logical functions NOR and NAND and have to be combined by the AND function \([2 \ 2 -3]\) in the third perceptron.

This combination represents the XOR function correctly, but leaves us with the problem how a net with more than two layers can be trained. We return to this question in the chapter on MLP networks.

### 3.6.3 Alternative output functions

Another objective is to solve the XOR problem by choosing an output function (in MATLAB: transfer function) different from the step function.

The most popular output function for neural networks is the sigmoid function (in MATLAB: logsig):
Unfortunately replacing the step function by the sigmoid function merely results in a continuous transition between the outputs 0 and 1 and leaves us with the same problem representing XOR.

We get an entirely different situation with the gauss function as output function:

Let us take the perceptron with weight/bias vector $[1 \ 1 \ -1]$ but the gauss function as output function:

Then the inputs $[0 \ 1]$ and $[1 \ 0]$ propagate activity 0 and hence output 1, the inputs $[0 \ 0]$ and $[1 \ 1]$ however propagate activity -1 or +1 and hence an output close to 0. This example is a good approximation of XOR but again we are left with the question, how such a network can be trained properly. We return to this problem in the chapter on RBF networks.

Concluding this section we see, that the perceptron approach is a good introduction to neural networks, but it is much too simple to be of use in more complicated situations. In fact in their book “Perceptrons (Oxford, England: M.I.T. Press. (1969))” Marvin Minsky and Seymour Papert gave a strict analysis of perceptrons extending the above arguments to arbitrary input dimensions. They showed that each perceptron divides its input space into two subsets which can be separated by a linear subspace. Hence a problem solvable by a perceptron must be linearly separable but for this kind of problems very efficient classical methods are available.
For about 20 years neural networks have been neglected by the public until new types of networks and training algorithms had been developed. These new networks had more promising features than the perceptrons and they revitalized the research in the field of artificial neural networks. Some of these Networks will be presented in the subsequent chapters.

3.7 Perceptron in MATLAB

We now want to use MATLAB to construct and train a perceptron. The reader should run the MATLAB system and try out the instructions given in this chapter.

3.7.1 Relevant MATLAB commands

```matlab
net = perceptron;
```

This command creates an empty perceptron in the typical perceptron topology:

1 input layer
1 processing layer with the hard limiter transfer function
1 output layer

The constructed net has various attributes, among others:

- `.inputs` with sub_attribute `.size`
- `.layers` with sub_attribute `.size`
- `.outputs` with sub_attribute `.size`
- `.IW` % the transfer matrix
- `.b` % the bias

Where `size` is the number of neurons in the layer under consideration.

Each of these attributes is a 1x1 cell array containing a single value or a matrix of values. A MATLAB net is a rather complicated structure which we further investigate in the upcoming chapters. The attribute `.outputs.size` has the special property of being read only, because the output sizes are completely determined by the topology and the other layer sizes and hence are automatically calculated.

Among others each MATLAB net has several methods:

- `net` (carrying the same name as the net itself) which takes a matrix of arbitrary many inputs and returns the vector of its net outputs.
- `view` displaying a view of the network structure.
- `train` takes a net, a matrix of inputs, and a vector of outputs and returns a net trained on this input-output-behaviour. As a side effect the method `train` adjusts the sizes of layers according to the needs of the given input and output.
- `save` takes the name of a file and the name of a variable to save the variable into the file for later use by a `load` command.
3.7.2 The definition of a perceptron

The MATLAB command to construct a perceptron (named `or_perc` here) is

```matlab
or_perc = perceptron;
view(or_perc);
```

using the view command we get a picture of the perceptron in MATLAB style:

![Figure 3-13: view of an empty perceptron](image)

We see here a perceptron where no neurons are placed yet, but the topology and output function is already prepared. We have an input layer (green) connected to the processing layer which has the step function (Hard Limit, `hardlim`) as output function (in MATLAB: `transfer function`).

![Figure 3-14: hardlim function](image)

The processing layer has the weight matrix `w` and a bias vector `b` (both empty so far). The output is transferred to the output layer without further processing.

There are two different ways of placing neurons into the different layers. The simplest is to create an input and target matrix for training. The training command then will first place the neurons into the layers and then do the training.

```matlab
x = [0 0 1 1; 0 1 0 1]; %input for binary logical function
t = [0 1 1 1]; %target for the OR function
or_perc = train(or_perc, x, t); %perform the perc.-training
```

This perceptron is now trained, neurons are placed into the layers as needed, and the weights are such that the perceptron is a model of the OR function. We check this by applying `or_perc` like a function:

```matlab
y = or_perc(x); % produces y = [0 1 1 1] the target
view(or_perc); %view the changed perceptron
```
PERCEPTRON

Figure 3-15: view trained perceptron
We can now read the weights and bias from the perceptron or_perc:

```
or_perc.IW{1};  % produces ans = [1 1] the actual weights
or_perc.b{1};   % produces ans = [-1] the actual bias
```

Use the save command to save the perceptron for later use:

```
save( 'OR_trained.txt' , 'or_perc')
```

A second way is to define the neurons directly without training:

```
or_perc = perceptron; % make or_perc an empty perceptron
or_perc.inputs.size;  % produces ans = [0] no input neurons yet
or_perc.layers.size;  % produces ans = [0] no neurons in the layer yet
or perc.layers.size;  % produces ans = [0] no layer has any neurons
```

Now we set the appropriate sizes:

```
or_perc.inputs.size = {2}; % 2 neurons into the Input
or_perc.layers.size = {1}; % 1 neuron into the layer
% the size of output is read only
or perc.layers.size; % produces ans = [1] so it is set automatically
view(or_perc); % view the changed perceptron
```

view produces the same picture as Figure 3-15 above, so we have the correct number of neurons in the layers. This implies that the matrix w and the bias b have the correct dimension; but their values are initialized by zeros:

```
orperc.IW{1}; % produces ans = [0 0]
orperc.b{1};  % produces ans = [0]
```

We now set the values in w and b “by hand”.

```
orperc.IW{1} = [2 2]; % both weights set to 2
orperc.b{1}  = [-1]; % bias set to -1
```

Now finally we have achieved some perceptron and check its performance:

```
y = or_perc(x);
% produces y = [0 1 1 1], again the target
We thus have constructed two perceptron models for the logical OR and save it for further use.

```matlab
save( 'OR_handmade.txt', 'or_perc' )
```

### 3.7.3 Visualization

Instead of only looking at the performance of the perceptron on the four 0-1-pairs, we get a better impression if we generalize to its behavior on the whole square.

```matlab
[R,S] = meshgrid(0 : 0.05 : 1 , 0 : 0.05 : 1);
```

creates a matrix of points in the square of the interval \([0,1]\) with the distance of 0.05 between neighboring points.

Now we calculate the net-outputs over the points of this meshgrid:

```matlab
T=R;
for i=1:size(R,1)
    T(i,:) = or_perc([R(i,:);S(i,:)]);
end;
```

Finally we plot the mesh over this grid:

```matlab
mesh(R,S,T);
```

Then we do the same thing for the trained or_perc by first loading:

```matlab
load( 'OR_trained.txt', 'or_perc' )
```

We add colored points as marks for the training input with its target and obtain two slightly different graphics.

![mesh/untrained](Figure 3-16)

![mesh/trained](Figure 3-17)

Observe that the trained perceptron has two of the red points (output 1) immediately on the border line between red and blue (output 0) while our own example has none of the points on this borderline. This shows that our "hand-
made” model has a better stability. For the trained model a minor deviation from the input [0 ; 1] or [1 ; 0] can change the output to 0 (blue) instead of the demanded 1 (red). This unstable behavior does not occur in the untrained model. We see in this simple example how a visualization of the model can help to compare different models of the same training task.

In later chapters we will investigate, how we can force higher stability by alterations of the training process or the transfer functions.
CLASSIFYING NETWORKS

CHAPTER 4:
MULTI LAYER PERCEPTRON

Goals
1. Feed forward networks and layered networks
2. grouping networks in MATLAB layers
3. gradient descent methods
4. back propagation
5. the black box
6. variations of back propagation
This chapter focuses on the most used type of neural networks, the so-called “multi layer perceptrons” (MLP) with their training rule “backpropagation”. As the name suggests, these networks are generalizations of simple perceptrons as presented in the last chapter, which fortunately are not so limited in their application.

The multi layer perceptrons differ from simple perceptrons in several ways:

**Output function:** the step function usually is replaced by the so-called sigmoid function, but also other functions are possible, preferably derivable functions.

**Topology:** the topology allows for arbitrary many layers of neurons instead of a single one for the simple perceptron, but it does not allow any backward connections.

**Training:** like in simple perceptrons the training is based on the delta rule which is applied layer by layer starting from the output layer going backwards towards the input layer.

**View:**

![Figure 4-1: a small MLP](image1)

![Figure 4-2: MLP in MATLAB view](image2)

There will be a slight distinction between **layers** and **levels** of neurons. Not many books make this distinction, but because of the use in MATLAB we have to distinguish between levels (neurons having the same distance from input neurons in the topology) and layers (groups of neurons on the same level). In this sense any level of neurons may be subdivided into several layers.

Also there is a distinction between feedforward networks (net without feedback) and strongly leveled networks (connections only exist from any level to the next higher level of neurons).
4.1 Feedforward networks

A feed forward network is defined as a network without feedbacks. In the sequel we use the more neutral term unit for either a single neuron or a group of neurons.

To be precise: In a feed forward network we can number the units in such a way that each connection leads (forward) from a unit to another unit with greater number and no connection leads (backward) from some unit to the same unit or a unit with smaller number.

A simple way to construct a feed forward network is to stepwise add a new unit and then define connections from units constructed before to that new unit (and no others). Then numbering the units in the sequence of their creation the connections can only go from a smaller number to a greater one.

4.1.1 Mathematical Excursion: Graphs

This section can be skipped on first reading, since it will only handle the problem to decide, whether a given network is feed forward and how to subdivide it into levels for better drawing and viewing.

Mathematically speaking a network is a directed graph which consists of units and connections, where each connection is determined by a pair u->v of units; u is the start and v is the end of this connection. A path of length n>0 through the graph is a sequence u_0 -> u_1 -> u_2 -> ... -> u_{n-1} -> u_n of n+1 connected units, and if the two units u_0 and u_n happen to be equal then the path is called circular or a cycle. No matter how we enumerate the units of a cycle at least one of the connections has to lead from a higher number to a lower number. Hence a graph is feed forward if and only if it contains no cycle.

There is a simple algorithm to construct the numbering of a feed forward network:

1. Start with number n=1.
2. Repeat:
   Search for a unit, which has no number yet and is not the end of a connection from any unit without number, give it number n and set n=n+1; until no such unit can be found.
3. If all units have a number the network is feed forward, otherwise the unnumbered part contains all the feedback cycles.
Figure 4-5: A Autobahn Crossing

The red circles on the junctions in this map are the units of this graph and the connections mark the possible direct route to a next junction. The existing connections are listed in the black rectangle. Now we apply the above algorithm:

The first numbers are given to those units, which are not end of any connections, hence A C G I are numbered 1 2 3 4 (any sequence possible). Now we search all units which are end only of connections with already numbered starts. M D O J are numbered 5 6 7 8.

Again we search all units which are end only of connections with already numbered starts. E P K are numbered 9 10 11.

Again we search all units which are end only of connections with already numbered starts. B F L N are numbered 12 13 14 15.

Again we search all units which are end only of connections with already numbered starts. H is numbered 16.

Now all units are numbered and by construction connections only go from lower to higher numbers.

The 5 groups of units we constructed shows that we have a longest path from units in group 0 to units in group n (= 1,2,3,4) of length n and we call these groups
MULTI LAYER PERCEPTRON

Levels or sometimes also layers of the graph. Since we use the notion layer in a different meaning in MATLAB we prefer the notion level in this lecture even though most neural network papers use the notion layer for this situation.

It is always possible to subdivide each feed forward network into subsequent levels, where a level is defined by the maximal number of steps from the input to the members of the level. We use the similar algorithm as above but increase the number after each group rather than after each unit:

1. Start with number k=0.
2. Repeat:
   a) take all units not in a level and which are not the end of connections from any units without level, put these into level k and set k=k+1;
   until a level remains empty
3. The levels only contain the units not contained in a feedback cycle.

The above example generates the levels:


Note that in level 1 there are only units having no connections from another unit in the net, hence level 0 contains the input units. In level k we find those units, which can be reached from the input level only in less than k+1 steps (consecutive connections).

Figure 4-6: Map as leveled network

We call a network strongly leveled if each connection leads from one level to the next higher level, hence it doesn’t jump across other layers as in the above figure. A consequence is, that all units in level k are reached from the input level in precisely k steps.

Let us take another crossing map to see what happens in the feedback case:
The above algorithm produces only the level 0: A C E G because all successors of A C E or G respectively are successors of I O M or K as well and hence the algorithm stops because all subsequent units are contained in the cycle I -> J -> O -> P -> M -> N -> K -> L -> I running through the clover leaf.

One could try to make groups as follows:

1: J P N L  2: O M K I  3: D F H B  but then connections will go from group 0 to 1 and 3, 1 to 2, 2 to 3, and also back from 2 to 1 creating the cycle above.
The difference between the two maps of Autobahn crossings is that in the second example we could get trapped in a cycle (clover leaf) never leaving the grossing again; in the first example this can’t happen because we leave the crossing after at most 4 steps.

Note that if we killed one of the connections between level 1 and 2 or 2 and 1 the cycle would be broken and network would fold up into a feed forward net with many more levels.

We leave now this excursion and return to the feed forward neural networks. In the XOR Example we have seen that a change of topology or a change of the output function can lead to a solution. We first look at the most common output functions, on the most important topologies, and on the implementation in MATLAB. Finally we add a chapter on the training algorithm.

4.1.2 Output functions

The standard output function (in MATLAB transfer function) for the perceptron was the step function which in MATLAB is called hardlim.

4.1.2.1 Hard Limit

\[
\text{hardlim}(n) = \begin{cases} 
  1 & \text{if } n \geq 0 \\
  0 & \text{otherwise}
\end{cases}
\]

We can draw this function via the MATLAB instruction:

\begin{verbatim}
  n = -5:0.1:5;  % 101 x-values between -5 and +5
  a = hardlim(n);  % the 101 function values
  plot(n,a)  % the plot command
  xlim([-5.05 5.05])  % x-axis range
  ylim([-0.05 1.05])  % y-axis range
\end{verbatim}
In the previous chapter we saw how the hardlim function works on the whole rectangle \([0, 1] \times [0, 1]\) with two perceptron models for the logical OR.

4.1.2.2 Identity

\[\text{purelin}(x) = x\]

We can draw this function via the MATLAB instruction:

```matlab
n = -5:0.1:5;  \% 101 x-values between -5 and +5
a = purelin(n);  \% the 101 function values
plot(n,a)  \% the plot command
xlim([-5.5 5.5]) \% x-axis range
ylim([-5.5 5.5]) \% y-axis range
```

We can see how the purelin function works on the whole rectangle \([0, 1] \times [0, 1]\) by changing the transfer function in the perceptron:

```matlab
load( 'OR_handmade.txt' , 'or_perc' );
\% load or_perc
or_perc.layers{1}.transferFcn = 'purelin';
```
% change the transfer function to purelin
save( 'OR_purelin.txt' , 'or_perc' );    % save it
[R,S] = meshgrid(0 : 0.05 : 1 , 0 : 0.05 : 1);
T=R;
for i=1:size(R,1)
    T(i,:) = or_perc([R(i,:);S(i,:)]);
end;
mesh(R,S,T)

Figure 4-11: purelin mesh

4.1.2.3 Sigmoid function

\[
\text{logsig}(x) = \frac{1}{1 + \exp(-x)}
\]

We can draw this function via the MATLAB instruction:

\[
\begin{align*}
n &= -5:0.1:5; \quad \text{% 101 x-values between -5 and +5} \\
a &= \text{logsig}(n); \quad \text{% the 101 function values} \\
\text{plot}(n,a) \quad \text{% the plot command} \\
\text{xlim([-5.5 5.5])} \quad \text{% x-axis range} \\
\text{ylim([-0.05 1.05])} \quad \text{% y-axis range}
\end{align*}
\]
Extend this to the square:

```matlab
or_perc.layers{1}.transferFcn = 'logsig';
[R,S]=meshgrid(-2 : 0.2 : 2 , -2 : 0.2 : 2);
for i=1:size(R,1)
    T(i,:) = or_perc([R(i,:);S(i,:)])
end;
mesh(R,S,T)
```

Figure 4-12: logsig

4.1.2.4 Radial basis function

\[ \text{radbas}(x) = \exp(-x^2) \]

We can draw this function via the MATLAB instruction:

```matlab
n = -5:0.1:5; % 101 x-values between -5 and +5
a = radbas(n); % the 101 function values
plot(n,a) % the plot command
xlim([-5.5 5.5]) % x-axis range
ylim([-0.05 1.05]) % y-axis range
```
Extend this to the square:

```matlab
or_perc.layers{1}.transferFcn = 'radbas';
[R,S]=meshgrid(0:0.05:1,0:0.05:1);
for i=1:size(R,1)
    T(i,:) = or_perc([R(i,:);S(i,:)]);
end;
mesh(R,S,T)
```

There are many more transfer functions as defaults in MATLAB and even more can be defined by the users. In this chapter we stick to these examples since they are widely used for feed forward neural networks.

### 4.2 Multi Level Topology

We have seen above that strongly leveled networks are a special case of feed forward networks, but since multi layer perceptrons are almost always strongly leveled, we will only handle those networks in this chapter. We return to the more general case only in later chapters.
MULTI LAYER PERCEPTRON

Also the multi layer perceptrons almost exclusively are equipped with the sigmoid function (‘logsig’ or ‘tansig’, where \( \text{tansig}(x) = 2 \cdot \text{logsig}(x - 1) = \tanh(x) \)), or purelin) so we mainly stick to these transfer functions during this chapter.

MATHLAB allows a very quick construction of multi layer perceptrons:

\[
\text{mlpnet} = \text{feedforwardnet}([2 \ 3 \ 6]);
\]

\[
\text{view(mlpnet)}
\]

These commands produce the view

Figure 4-16: \text{mlp([2 \ 3 \ 6]) unconfigured}

As we can see the vector \([2 \ 3 \ 6]\) produces 3 hidden layers (these are the layers which are not output-layers = layers with connection to output) with the sizes 2, 3, 6 and with default transfer function ‘tansig’, the output layer has default transfer function ‘purelin’, that means the output layer just produces the output and transfers it unchanged. Input and output have default size 0 (empty); we say the net is \textbf{unconfigured}.

In general

\[
\text{net} = \text{feedforwardnet}([x_1 \ x_2 \ldots x_n]);
\]

returns a feed forward net with \(n\) hidden layers of sizes \(x_1, x_2, \ldots, x_n\) with transfer function ‘tansig’ as default. The net is unconfigured, i.e. input and output have size 0.

When starting a training process the first step is configuring the network according to the dimensions of input and output using an internal command \texttt{configure}. We also can do the configuration by hand (compare the perceptron \texttt{net = perceptron}; in the last chapter):

\[
\text{mlpnet.inputs.size} = \{5\}; \quad \% 5 \text{ input neurons}
\]

\[
\text{mlpnet.layers.size} = \{2; 3; 6; 4\}; \quad \% 4 \text{ output neurons, hidden layers unchanged}
\]

\[
\text{view(mlpnet)};
\]

Figure 4-17: \text{mlp([5 \ 2 \ 3 \ 6 \ 4]) default}

We also can change the default transfer functions ‘tansig’ and ‘purelin’:
MULTI LAYER PERCEPTRON

```matlab
mlpnet.layers.transferfcn={'logsig';'logsig';'logsig';'logsig'};
view(mlpnet);
```

**Figure 4-18**: mlp([5 2 3 6 4] 'logsig')

There is no essential difference in the network performance between using 'tansig' or 'logsig', the latter however will make the analysis of the net behavior a bit easier. We come to this point in a later chapter.

We have now constructed a MLP (multi layer perceptron) named `mlpnet` which expects as input a 5×1-matrix (column vector) and returns a 4×1-matrix as output.

```matlab
mlpnet([1;1;1;1;1]) % under default weights produces
ans =
0.5000
0.5000
0.5000
0.5000
```

MATLAB allows to apply the network to many inputs in parallel, i.e. the input may be an arbitrary 5×n-matrix producing a 4×n-output-matrix.

```matlab
mlpnet([1 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1])
ans =
0.5000 0.5000 0.5000 0.5000 0.5000
0.5000 0.5000 0.5000 0.5000 0.5000
0.5000 0.5000 0.5000 0.5000 0.5000
0.5000 0.5000 0.5000 0.5000 0.5000
```

4.2.1 Weights and biases

So far we have not initiated any weights, so they have their default values by construction. Default values are either zeros or random values which we can read from the `IW LW` and `b` properties of the network.

```matlab
A = mlpnet.IW
B = mlpnet.LW
C = mlpnet.b
```

produces 3 cell-matrix variables which can easily be explored by the variable editor of MATLAB.
A must be interpreted that only the first layer (with 2 neurons) has a connection from the input (with 5 neurons) and hence we have a $2 \times 5$-matrix as input weight matrix with default zero.

B is interpreted: the only connections are into layer 2 from layer 1, into layer 3 from layer 2, and into (the output-) layer 4 from layer 3. These connections are weighted at random except for the last matrix in the output layer with default zero.

Finally C is interpreted: all 4 layers have a bias for every neuron, where again the biases for input and output layer are default zero and the others at random.

This strange mixture of default values has its reason in the two step construction of mlpnet. The first construction step with feedforwardnet creates the hidden layers in their actual size with weights and biases chosen at random. In the second step the input and output neurons were created by assigning new sizes to the input and output, and thereby creating the weights and biases by zero default. If we had changed the size of any of the hidden layers, also the newly created weight and bias values would have had default zero.

The interpretation above also shows that there is a distinction between “no connection” and “connection with weight 0”, even though replacing an empty matrix (no connection) by a zero matrix would not have any effect on the input-output-behavior of the network. The distinction arises during a training process, where a connection of weight zero might change its weight to a nonzero value and thereby influence the output. If there is no connection between some pair of neurons (or layers) no (weight-changing) training process can suddenly create a connection between those neurons. 

Every change of weights and biases produces a new input-output-behavior of the network, hence we can change the quite uninteresting behavior of our example mlpnet (constant output 0.5) by redefining the values in the different weight and bias matrices altogether $2^5 + 3^2 + 6^3 + 4^6 + 2 + 3 + 6 + 4 = 73$ values.

Obviously this quite innocent looking example already leads to a 73-parameter problem: which weight/bias allocation leads to a desired input-output behavior of the network? It is very hard to solve this problem by hand or by experiment because even if we allow say only 10 values for each parameter we end up with $10^{73}$ different possible parameter allocations which we have no chance to completely investigate, even with high speed computers (curse of dimensionality).

---

4 It is conceivable to design a connection creating training process, but we will not consider this kind of training here.
Remark:
10^{67} is the order of magnitude of the number of Atoms in our galaxy, so we have no chance to save all these parameter sets.
10^{73} nanoseconds = 10^{64} seconds = 3 \times 10^{56} years, this exceeds the lifetime (10^{20} years) of the universe by many orders of magnitude, so we have no chance to go through all possible parameter sets for only a nanosecond each.
We have no chance for a lucky guess for the optimal solution since it is as probable as winning a 6 numbers out of 49 lottery game with the same number combination 10 times in a row.
(\text{compare } \text{http://fma2.math.uni-magdeburg.de/~bessen/krypto/krypto8.htm})

The only way to solve the problem of finding a suitable parameter set can be a mathematical approximation method to come close to an optimal solution. Any such method must postulate some continuity conditions on the problem to be really effective. The very least assumption is that any two parameter sets with small distance have very similar performance. This observation is the reason why in general with MLP we stick to continuous (even continuously differentiable) transfer functions for all neurons.

4.2.2 Backpropagation training

In this section we investigate a special class of \textbf{supervised} training algorithms i.e. the training is controlled via a series of input data sets \textbf{X} and a similar series of corresponding output data sets \textbf{T} (the so called \textbf{targets}). The aim is to construct a functional network model \textbf{net} with the input-output-behavior

\[ \textbf{net}(\textbf{X}(i)) = \textbf{T}(i) \]  
for all indices \textit{i} of the series.

Because of the curse of dimensionality it would be too optimistic to expect an exact solution therefore the training process should be oriented at an error limit which is not to be exceeded. Mathematically speaking training is an approximation process in a (weight/bias-) parameter-defined family of networks searching for a model with error below the limit.

One of the most prominent approximation methods is \textbf{gradient descent}. Here the error is regarded as function of the parameters and for a given parameter set the error gradient is calculated and a multiple of the gradient is added to the parameter set to get the next parameter set. Since the gradient marks the direction of the steepest descent of the error function, a good choice of the step length (the above “multiple”) leads to a parameter set with smaller error.

Illustration:
If a hiker is caught in fog in the mountain he can only see a small circle around him, just enough to see which direction the way goes downwards. In order to get down from the mountain he will try to do small steps always in down direction to finally arrive in the valley.
Unfortunately only taking small steps the hiker may get into a small pit still way up in the mountain; since from there all directions lead upwards the algorithm does not work anymore and the hiker is stuck in the fog.
This illustration shows that gradient descent can lead us to parameter sets causing ever smaller errors, however if we step into a point of gradient zero then we have no direction in which the error decreases (this is called a local minimum) and the algorithm stops. In this case we might still have too large errors to call this approximation process successful.

Another problem might be that we find ourselves in a situation with gradient very close to zero (this is called a plateau) which still may be on very high error level, but too many steps may be needed to reduce the error substantially.

The gradient descent method is an effective mathematical method to do parameter driven approximations by reducing the error stepwise. The algorithm ends whenever a gradient zero is reached, then the error has reached a local minimum, which may be bigger than the global minimum (the minimal theoretically achievable error value).

In this section we see that perceptron learning is a special case of gradient descent and it is easily extendable to multi layer perceptrons.

### 4.2.2.1 Perceptron learning

Let us look back to the previous section, where we investigated perceptrons. As the only change we replace the step function by the linear function purelin.

```matlab
net = feedforwardnet([]);
net.inputs.size = {3};
net.layers.size = {2};
view(net);
```
Out = net([x1 ; x2 ; x3]) = [w1*x1 + w2*x2 + w3*x3 - b]

err([x1 ; x2 ; x3]) = (net([x1 ; x2 ; x3]) - target)

The gradient of the output is

\[ \text{grad}(\text{err}) = [\partial \text{err}/\partial w1 \ \ \partial \text{err}/\partial w2 \ \ \partial \text{err}/\partial w3 \ \ \partial \text{err}/\partial b] \]

which calculates to

\[ [x1 \ x2 \ x3 \ -1] \]

and hence the perceptron learning rule says, that the weight/bias vector has to be changed by

\[ \Delta w = \text{err}*\text{grad}(\text{err}). \]

If we had used the sigmoid function \( \logsig \) instead of purelin, we would have to multiply each component of the gradient by the internal derivation \( \logsig'(Out) \). Since \( y=\logsig(x) \) has a very particular derivation \( \logsig'(x)=y*(1-y) \), we end up with the formula

\[ \Delta w = \text{Out}*(1-\text{Out})*\text{err}*\text{grad}(\text{err}) \]

In any case the change of the weights/bias is a multiple of the gradient and so in fact the perceptron training rule is a special instance of gradient descent.

Note that if we had taken the quadratic error \( \text{err} = (\text{Out} - \text{target})^2 \) the gradient would have been \( \text{grad}(\text{err}) = \text{err}^*\text{grad}(\text{Out}) \) and hence

\[ \Delta w = \text{grad}(\text{err}) \text{ or } \Delta w = \text{Out}^*(1-\text{Out})*\text{grad}(\text{err}). \]

### 4.2.2.2 Multi level gradient descent

We now want to derive a simple formula for gradient descent for a multi level topology. For this lecture it is not important to learn this formula or its derivation, the aim is only to understand the structure of this and similar training methods and why this training method is called “backpropagation”.

Let us look at an arbitrary feed forward net  

\[ 5 \ \logsig(x)= 1/(1+exp(-x)) , \]

\[ \logsig'(x)=\frac{-1/(1+exp(-x))^2)*exp(-x)^*(-1)}{\frac{-1+1+exp(-x))/(1+exp(-x))^2}{\frac{-1/(1+exp(-x))^2}+(1/(1+exp(-x))} = \logsig(x) - \logsig(x)^2 . \]
This net expects an input (column-) vector $x_0$ of length $k_0$ each Layer number $l=1,\ldots,n$ produces its output vector $o = x_l=\theta(a)$ of length $k_l$, where the activity $a = W*u - b$ and $\theta$ is the transfer function (e.g. logsig) and $u = x_{l-1}$ is the layer input; finally the net-output is $o = x_n$ and the errors at the different components of $x_n$ are the components of $e_n = x_n-target$. These errors can be combined to a single error value e.g. by the quadratic error $err = (x_n-target)^2 = (x_n-target)^t*(x_n-target)$.

In the sequel we calculate the gradients $\frac{\partial err}{\partial W}$ for all layers $l=1,\ldots,n$ by calculating the intermediate errors $\frac{\partial err}{\partial o}$ of the layer outputs $o=x_l$ in between. In his calculations we will need as further intermediate steps the derivation of the activity function $\theta'(a)$ which is the column vector $[\theta'(a_1); \theta'(a_2); \ldots ; \theta'(a_k)]$.

We calculate the errors and gradients backwards for $l=n,\ldots,1$ and for a better readability we do not equip the matrices $W$, the inputs $u$, the biases $b$, the activities $a$, and the outputs $o$ with the corresponding layer indices $l$. So in every layer $l=n,\ldots,1$ we have the formulae

$$x=\theta(a) \quad a = W*u - b.$$ 

For $l=n$ and $i=1,\ldots,k$:

$$\frac{\partial err}{\partial o} = (o-target)^2/\partial o = 2*(o-target)$$

For $l\leq n$:

$$\frac{\partial err}{\partial u} = (\frac{\partial err}{\partial o})*\frac{\partial o}{\partial u}$$

$$= (\frac{\partial err}{\partial o})*\theta'(a)*\frac{\partial a}{\partial u}$$

$$= (\frac{\partial err}{\partial o})*\theta'(a)*W$$

For simplicity we abbreviate the product $(\frac{\partial err}{\partial u})*\theta'(a)$ in layer $l$ by $e_l$ and thereby obtain the recursive formula for the intermediate errors:

$$e_n = 2*(x_n-target)*\theta'(a_n)$$

$$e_{l-1} = e_l*W*\theta'(a_{l-1})$$

Note that this recursion formula states that the error vectors $e_l$ are calculated by propagating the output error $e_n$ backwards from layer to layer by multiplying with the same Weight-matricies $W$ used for the propagation of the input towards the net output. For this reason this mechanism is called
backpropagation (or more precise: backpropagation of errors).
Now we turn to the calculation of the gradient.
\[ \frac{\partial \text{err}}{\partial W_{ij}} = \sum_t \left( \frac{\partial \text{err}}{\partial u_t} \right)^* \left( \frac{\partial u_t}{\partial W_{ij}} \right) \]
\[ = \sum_t \left( \frac{\partial \text{err}}{\partial u_t} \right)^* \theta'(a_t)^* u_t \]
\[ \frac{\partial \text{err}}{\partial b_i} = \sum_t \left( \frac{\partial \text{err}}{\partial u_t} \right)^* \left( \frac{\partial u_t}{\partial b_i} \right) \]
\[ = \sum_t \left( \frac{\partial \text{err}}{\partial u_t} \right)^* \theta'(a_t)^* \left( \frac{\partial a_t}{\partial b_i} \right) \]
\[ \frac{\partial a_t}{\partial W_{ij}} = 0 \text{ if } t \neq i \]
\[ \frac{\partial a_t}{\partial b_i} = -e_i \text{ if } t = i \]
The calculation shows, that we get a recursion in each layer:
\[ \frac{\partial \text{err}}{\partial W} = e^* u^t \]
\[ \frac{\partial \text{err}}{\partial b} = -e \]
Combining these calculations we see that for each Layer we obtain the Δ-Rule
\[ W_{\text{new}} = W_{\text{old}} + \eta * e^* u^t \]
\[ b_{\text{new}} = b_{\text{old}} - \eta * e \]
for the weight changes and bias-changes in that layer \( l \), where \( e = e_l \) is calculated by the above backpropagation recursion. The value \( \eta \) is the so called step width for the learning step, it controls which multiple of the gradient the weight changes should be. If the transfer function \( \theta = \text{logsig} \) the backpropagation formula simplifies even more since \( \theta'(a) = \theta(a)^*(1-\theta(a)) = x.*(1-x) \) and so no explicit calculation of \( \theta' \) is necessary.
\[ e_n = 2*(x_n-\text{target})*x_n.*(1-x_n) \]
\[ e_{i-1} = e_i^*W^*x_{i-1}.*(1-x_{i-1}) \]
We have now learned how a single learning step in a multi level network could work, but there are some more things to keep in mind before we obtain a full training algorithm. In general a training algorithm has several separated calibrations to perform:

4.2.2.2.1 net dimensions
The dimensions of the layers are set during the net construction as explained above, thereby the transfer functions, weights and biases are initialized by default or in addition by hand.

4.2.2.2.2 training data
The function train is designed to divide the input file into 3 subfiles \( \text{TrainV}, \text{ValV}, \) and \( \text{TestV} \) according to the net's divideFcn (default: \text{dividerand} = divide at
random) and its `divideParam` with the subproperties `trainRatio`, `valRatio`, and `testRatio`, which tell what percentage of the input data should fall into the Train-, Validation-, or Test- Dataset, the default Ratios are 0.7, 0.15, 0.15. The subdivision of the input data into subfiles can be turned off by the command `net.divideFcn='';` then all available input data are taken for the training.

4.2.2.2.3 online/offline

The decision, whether the training is to be performed in batch mode (i.e. offline: the weight/bias changes are accumulated until all training data have been presented and then the accumulated change is performed) or alternatively in incremental mode (i.e. online: after each presentation of training data a weight/bias change is performed) is done by presenting the training data either as a matrix e.g. \([1 \ 2 \ 3; 2 \ 1 \ 3] \) or alternatively as a cell array e.g. \({[1;2] \ [2;1] \ [2;3] \ [3;1]} \). In the first case the weight/bias change is performed after all (4) inputs have been presented, in the second case the change is made after every single input.

4.2.2.2.4 Stopping

As we said before there are several reasons, why training could fail, but in most cases the algorithm would go on forever without producing any better results. For this reason the algorithm has several stopping conditions controlled by the net’s `trainParam` values. Training stops when any of these conditions occurs
- **epochs**: the maximal number of `epochs` (= runs through `TrainV`) is reached.
- **goal**: the performance (total error) of the net falls below the goal
- **min_grad**: the gradient falls below `min_grad` (i.e. the algorithm is caught in a local minimum and hence must be stopped)
- **max_fail**: the validation performance (=error on `ValV`) has increased more than `max_fail` times since the last time it decreased (in case validation is used)

4.2.2.2.5 learning control

`lr`: is the learning rate (the above factor \(\eta\) controlling the step width)
for normal gradient descent `lr` is constant and controls the learning speed, however more complex variations of gradient descent may have further parameters influencing the value of the variable `lr`. 
4.2.2.6 learning feedback

**TR:** the training record contains all the settings and performance information during the training of the net. It is the second argument of the train function:

\[
[\text{net}, \text{tr}] = \text{train}(\text{net}, \text{inputs}, \text{targets});
\]

**show_commandline:** produce performance information on the command line always after **show epochs**

**show:** number of epochs after which performance information is displayed

**show-window:** plot performance information of the training in an extra window.

Using the learning feedback a **Post-Training Analysis (Network Validation)** has to follow in order to decide, whether the training has been successful. In case of a failure a decision has to be made, what changes in the training process could improve the result.

### 4.2.3 training variations

In the numerical mathematics many approximation algorithms have been developed based on simple gradient descent. Many of these methods have been reformulated into backpropagation learning rules resembling the rule derived above. The most important algorithms have been incorporated in MATLAB and can be seen in the following table

<table>
<thead>
<tr>
<th>Function</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>trainlm</td>
<td>Levenberg-Marquardt</td>
</tr>
<tr>
<td>trainbr</td>
<td>Bayesian Regularization</td>
</tr>
<tr>
<td>trainbfg</td>
<td>BFGS Quasi-Newton</td>
</tr>
<tr>
<td>trainrp</td>
<td>Resilient Backpropagation</td>
</tr>
<tr>
<td>trainscg</td>
<td>Scaled Conjugate Gradient</td>
</tr>
<tr>
<td>traincgb</td>
<td>Conjugate Gradient with Powell/Beale Restarts</td>
</tr>
<tr>
<td>traincfgf</td>
<td>Fletcher-Powell Conjugate Gradient</td>
</tr>
<tr>
<td>traincgp</td>
<td>Polak-Ribiére Conjugate Gradient</td>
</tr>
<tr>
<td>trainoss</td>
<td>One Step Secant</td>
</tr>
<tr>
<td>traingdx</td>
<td>Variable Learning Rate Gradient Descent</td>
</tr>
<tr>
<td>traingdm</td>
<td>Gradient Descent with Momentum</td>
</tr>
</tbody>
</table>
The most flexible among these algorithms is the Levenberg-Marquardt algorithm which therefore is used as default learning rule for feed forward nets in MATLAB. We do not go into any details of the different algorithms, since the development of learning processes is not in the focus of this lecture. For our purposes it is enough to know that there are some algorithms implemented in MATLAB from which we can freely choose reading/writing the `trainFcn` property of the net.

```
net.trainFcn = 'traingd'
```

replaces the default training function by the simple gradient descent training. Each training function has its own set of parameters controlling the training process e.g. the default training function `net.trainFcn = 'trainlm'` has instead of the constant learning rate \( lr \) an adaptive learning rate \( \mu \) with the following changing rules:

<table>
<thead>
<tr>
<th>Property</th>
<th>default</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.mu</td>
<td>0.001</td>
<td>Initial ( \mu )</td>
</tr>
<tr>
<td>net.trainParam.mu_dec</td>
<td>0.1</td>
<td>( \mu ) decrease factor</td>
</tr>
<tr>
<td>net.trainParam.mu_inc</td>
<td>10</td>
<td>( \mu ) increase factor</td>
</tr>
<tr>
<td>net.trainParam.mu_max</td>
<td>1e10</td>
<td>If ( \mu ) exceeds Maximum</td>
</tr>
</tbody>
</table>

The adaptive value \( \mu \) is increased by the factor \( \mu_{inc} \) until the change above results in a reduced performance value. The change is then made to the network and \( \mu \) is decreased by the factor \( \mu_{dec} \). Hence if the step width \( \mu \) already results in a reduced error it \( \mu \) not increased but after the change it is decreased. In all other cases at least one increase and one decrease occurs so that \( \mu \) stays equal or increases altogether. It is clear that changing the default values still \( \mu_{dec} < 1 \) and \( \mu_{inc} > 1 \) must hold. As a rule of thumb we can say that values close to 1 make the algorithm somewhat slower and values distant to 1 make it quicker.

The reason for this adaptive handling of the step width is that the Levenberg-Marquardt-algorithm is a 2\(^{nd}\) grade algorithm (calculating the Jacobi matrix instead of the gradient); such an algorithm is much faster than the ordinary gradient descent but much more vulnerable to get caught in local minima or plateaus. In this case increasing the step width may help to leave the local minimum or the plateau, but for normal use the step width should decrease again to normal size.
Figure 4-22: escape local minima

We see in this picture an example of a point on an error curve, where a short step does not lead to a smaller value in the error curve, a long step however may escape the local minimum and lead to a smaller error value. Another problem can arise, if the step width is too large and the error curve has a steep valley.

Figure 4-23: narrow valley

In this example the step width is such that the training algorithm jumps between two points back and forth which are located on opposite sides of a narrow valley. Only a shorter step would allow reaching a point lower down in that valley representing a smaller error.

These examples show that an intelligent adaption of the step width during the training can help the training to overcome the typical flaws of simple gradient descent.

4.3 Radial Basis Function Networks
Another variation is the radial basis network type which can be interpreted as a multi dimensional variation of the classical Gaussian approximation in Mathematics.

4.3.1 Gaussian Approximation

The background for another type of neural networks is the mathematical function approximation by superposition of Gaussian bell curves \( y = a \cdot \exp(-b \cdot (x-c)^2) \), whose standard version \( y = \exp(-x^2) \) in MATLAB is called radial basis functions and used as transfer function \( \text{radbas} \). The command \( p = -3:1:3 \); \( a = \text{radbas}(p) \); \( \text{plot}(p,a) \) produces a plot of \( \text{radbas} \):

![Gaussian Bell Curve](image)

**Figure 4-24 : Gaussian Bell Curve**

In general the bell function has a symmetry axis at \( x = c \) (\( c \) is called the center) and it has its maximum \( a \) (or minimum \( a \) if \( a < 0 \), \( a \) is called the amplitude) at \( x = c \). The output of the bell function depends on the distance \( |x-c| \) of \( x \) from the center. The role of \( b \) is to determine the breadth of the bell curve, finally by adding a constant \( s \) we can shift the whole curve up or down.

We get an example of a superposition of 3 bell curves, which is just a weighted sum of \( \text{radbas} \) functions with different centers and amplitudes.

\[
\begin{align*}
a_1 &= 0.5 \cdot \text{radbas}(p); \\
a_2 &= 0.6 \cdot \text{radbas}(p-1.5); & \text{amplitude 0.6 , shift right 1.5} \\
a_3 &= 1.1 \cdot \text{radbas}(p+2); \\
a_4 &= a_1 + a_2 + a_3; \\
\text{plot}(p,a_1,'b-',p,a_2,'b--',p,a_3,'g--',p,a_4,'m-') \\
\text{title('Superposition of Radial Basis Functions')}; \\
\text{xlabel('Input x')};
\end{align*}
\]
MULTI LAYER PERCEPTRON

ylabel('Output y');

produces a magenta curve which is the sum of the 3 bell curves.

We exemplify Gaussian approximation using the MATLAB-demo radial basis approximation which can be found in the file demorb.m.

Define 21 inputs P and associated targets T.

```
P = -1:.1:1;
T = [-.9602 -.5770 -.0729 .3771 .6405 .6600 .4609 ...
   .1336 -.2013 -.4344 -.5000 -.3930 -.1647 .0988 ...
   .3072 .3960 .3449 .1816 -.0312 -.2189 -.3201];
plot(P,T,'+');
title('Training Vectors');
xlabel('Input Vector P');
ylabel('Target Vector T');
```

Figure 4-25 : superposition of bell curves
We would like to find a function which fits the 21 data points.

The Gaussian approximation starts with a given number \( k \) of bell functions and optimizes the \( 3n+1 \) parameters (\( n \) centers, \( n \) breadths, \( n \) amplitudes, and 1 shift) to approximate the given points with the superposition of these \( n \) bell curves. In this simple example it seems obvious, that we can take a superposition of 3 bell functions with appropriate amplitudes centers and breadth and a suitable shift (9 parameters) to obtain a first satisfactory result.

\[
\begin{align*}
X &= -1:.01:1; \\
O &= \text{ones}(1,201); \\
a0 &= 0; \\
a1 &= -0.1*\text{radbas}(8*(X+0.05*O)); \quad \% \text{shift down} \\
a2 &= 1.7*\text{radbas}(3*(X+0.54*O)); \quad \% \text{fine tuning of the middle part} \\
a3 &= 1.4*\text{radbas}(2*(X-0.54*O)); \quad \% \text{representing the left hill} \\
d &= O + a1 + a2 + a3; \\
\text{plot}(P,T,'+',X,a1,'r-',X,a2,'g-',X,a3,'g-',X,d,'m-');
\end{align*}
\]
The success of an approximation is measured by the Euclidian distance (**mean square error MSE** or **root mean square error RMSE**) between the approximation output and the training data. The more initial bell curves we take the better the approximation can be, so if the approximation is still too imprecise we repeat the approximation process with an increased number of bell curves. We do not explain the approximation process in more detail here since this would go beyond the scope of this lecture.

### 4.3.2 RBF Networks

The concept of Radial Basis Function Networks has been developed by Chen et al. in 1991 as an extension of Gaussian approximation to arbitrary dimensions. This network type has two layers: a hidden layer of \( s \) **radbas**-neurons and an output layer with **dim(out)** **purelin**-neurons. The number \( s \) is the number of radial basis functions used for the approximation and the number of linear neurons is the output dimension.

A simple example can be obtained by the following MATLAB commands

```matlab
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P,T);
NEWRB, neurons = 0, MSE = 2.54
NEWRB, neurons = 2, MSE = 0
```

---


In a RBF Network the hidden layer works a bit different than the hidden layers of an ordinary MLP: instead of taking the matrix product of \( W \) with the input vector we take the vector Euclidean distances of the input to the rows of \( W \) and multiply elementwise with the bias vector (here: the breadth-vector) \( y_i = \text{radbas}(|x - W_i| \times b_i) \). The matrix \( W \) of the output layer contains the amplitudes for the bell functions, hence the output is the superposition of the bell functions from the hidden layer equipped with those amplitudes.

The training runs in several steps each of which has two parts. The first part determines the centers (rows of \( W \)) and breadth (biases \( b \)) of the bell functions, and the second determines the amplitudes (\( W \) in the output layer) and possible shifts (\( b \)) of each output variable.

In MATLAB a RBF network is constructed with a call `newrb(P,T)`, where \( P \) is an input matrix and \( T \) is the corresponding target matrix. As a consequence this call produces a sequence of RBF Networks with increasing number of neurons and its error (MSE) after a perceptron training in the output layer. The sequence ends, if the error stays below a predefined bound.

RBF nets work well for function approximation with low dimensional inputs, so it may not always be useful in high dimensional wind power prediction problems.

The main value is its new kind of approach by separately training a part of the net (here the RBF layer) solely on input data and only the remaining part (here the output layer) on output data.

**4.3.3 Input-output-separation**

A separation of input and output is particularly helpful in situations, where we have a whole wealth of input data but there is a deficit of input data coupled with corresponding target data. E.g. in wind power prediction we have an immense number of weather data from a whole maze of prediction points and dating from the past, but only very few of them are relevant to existing wind farms and corresponding wind power measurements.

For a network realizing input-output-separation we need a front part of the network being trained on the input data, without any reference to possibly existing target data. In a second phase the back part of the network is trained in an output oriented manner (such as back propagation) with those data equipped with corresponding target data.
MULTI LAYER PERCEPTRON

As an example we take a 50 point input set

![Figure 4-29: Example: input data](image)

In the first phase we select centers for the possible bell functions (e.g. 8 centers)

![Figure 4-30: selection of centers](image)

A selection of centers gives rise to a classification by greatest similarity on the input space, every possible input is classified to the center, which has the lowest distance to it. After having created this classification the second phase is to train the back part of the network according to the target data on each center:
Training only on the centers provides a very fast learning procedure, but may result in high error rates, where the centers are not chosen densely enough. To improve the performance of the net (reduce the error rate) it is necessary to run through the two phases several times and thereby choosing more and better distributed centers in the available input data.

The input-output-separation of RBF networks provides a first example of a combination of two different network types (classification and approximation) to one greater network. In the next chapter we will study classification networks in some more detail.
Goals
1. The concept of Competition in neural networks
2. Winner take all methods
3. The concept of self organization
4. Vector Quantization
5. Kohonen learning
6. Comparison of minimal Distance and maximal vector product

CLASSIFYING NETWORKS

CHAPTER 5: CLASSIFYING NETWORKS
In this chapter we consider a different type of networks, which are not designed to interpret fragmentary data sets as functions by interpolation; that means that the output at an unknown input is a compromise between the outputs of surrounding inputs. Why is anything else necessary? There are situations, where a compromise is the worst you can do. To obtain a demonstrative example we assume we are constructing an automatic steering device for a car.

![Figure 5-1: no compromise](image)

A car running towards a T-junction has two equally good possibilities, “turn left” or “turn right”, but any compromise between the two would be fatal. This little example shows, that the search for always a single correct answer for any question is not realistic, we must instead also allow for several useful answers to the same question. This easy example shows that, problems exist which cannot be fully described by a functional model.

Here we investigate an alternative type of model which answers the question: “which inputs behave similarly” rather than “what is the correct output for a given input”. These are networks like the front part of RBF Nets, which realize a classification (= clustering) of the input data rather than simulating a certain output behavior. Such networks are purely input oriented and no target vectors are necessary. A typical structure of these networks is a one layer topology like for perceptrons, the training strategy however is entirely different.

There are two basic ways to define a classification on a set of data, both begin with setting an initial number N of classes. This number may or may not change during the construction process:

- The first way is to define a function F from the input space into the number set 1..N (or into a set of N colors or ...), then two or more inputs belong to the same class if they are mapped to the same number by the function F (the kernel of the function).
The second way is to select $N$ elements in the input space as intended representatives (also called prototypes or centers) for the classes under construction and then define an input to belong to the class of the representative having the smallest distance to the input under consideration (classification by greatest similarity).

The first method is used, if for at least some of the inputs a classification is already predefined and the training has the goal to extend the existing classification to a wider set of inputs. This training is at least in parts a supervised learning and can be done with a back propagation like technique.

The second method has no predefined targets and hence training is unsupervised. The advantage of unsupervised learning becomes apparent if we try to model very large sets of inputs. Because of complexity there is no chance to subdivide the inputs into meaningful clusters unless there is some a priori information about the input set defining such a separation. So if as usual there is no a priori information an unsupervised clustering algorithm can provide some new insight into the structure of an otherwise confusing set of data.

After a clustering process the question arises how a clustering of the same data set should be rated better or worse than another. In particular the question is, when such a process should stop (what is “success” for a clustering). If we define the
diameter of a class as the maximum distance of members of that class, a stopping condition for the training algorithm could be that the diameter falls below a certain bound. Most algorithms however simply stop because of a limitation on the number of steps. An alternative stopping condition is defined by a step by step decreasing learning rate, which makes the algorithm stop if the learning rate reaches zero.

The training strategies in use are competitive in nature: the chosen centers compete with each other for the inputs and the winner gets the input in its class and changes its weight coordinates accordingly; this method therefore is called winner-take-all-strategy. Instead of determining the winner by greatest similarity it is possible to use alternative classifications such as k-nearest-neighbors, least-squares, k-means-clustering, and vector quantization. The discussion of these methods goes beyond the scope of this article. We will discuss two network types with a winner take all learning strategy.

5.1 Learning Vector Quantization (LVQ)

In the literature we find two definitions of LVQ networks, without or with an output-backend. The MATHLAB function `lvqnet` represents the latter version, where the hidden layer represents an arbitrary number of classes and the output layer combines these classes to the predefined classes of the training set (supervised learning). We describe here only the front-end version which is trained in unsupervised manner.

Our LVQ-network has only one layer whose neurons represent the classes. The lines of the weight matrix $W$ are the coordinates of the centers, there are no biases and all transfer functions are compet. This layer computes all distances of the input to the lines of $W$ and $\text{compet}$ returns a 1 if the line has minimum distance (winner) from the input and 0 else.

![Figure 5-4: LVQ learning](image)

If the $k$-th line $c_k$ is the winner, this center is moved a piece towards the input $x$:

$$c_k = c_k + \eta (c_k - x)$$

The effect of this learning process is that only the winner is moved a short way towards the input which is controlled by the learning rate $\eta$. After sufficiently many runs through the training data each center will be moved close to the midpoint of its class. Note that during the training process inputs may change.
their class since their centers are moving around. The learning rate \( \eta \) is reduced after each epoch (pass through the training set).

A supervised variant of this training process is illustrated in:

![Figure 5-5: supervised LVQ learning](image)

Here the same formula is used but the sign of the learning rate \( \eta \) changes depending on the correctness of the class found for the input. The classification results of this approach are not very convincing but the results of the MATLAB `lvqnet` are much better, since in the unsupervised training of the hidden layer there can be a much larger number of classes as parts of the fewer predefined classes in the output level.

![Figure 5-6: example lvqnet](image)

In this example e.g. the supervised LVQ rule would be bound down to the 3 possible classifications whereas `lvqnet` allows as many intermediate classes as necessary (10 in this case) to create a pre-grouping of the inputs and the groups are then combined to the 3 desired classes. Clearly such a model is much more flexible, than the pure supervised LVQ.

As we see below the unsupervised version of LVQ can be considered a trivial special case of SOM.

### 5.2 Self Organized Maps (SOM)

The concept of a **Self Organized Map** or **Self Organized Feature Map (SOFM)** has been developed by Teuvo Kohonen, so in the literature these networks are also addressed as **Kohonen Maps**. Their topology is very much like unsupervised LVQ-Nets and front-ends of RBF-Nets, in particular they have only one layer, which has `compet` as transfer functions and no biases. Instead of the scalar product SOM nets use the distance (greatest similarity).
MATLAB sometimes also uses the scalar product for the calculation of the winner by the minimal activity and we shall explain later, why this is no big difference from using the maximal distance like in RBF Nets.

The important extra that comes with a SOM is a 1-chain or a 2-dimensional grid topology defined on the output layer.

The grids can be rectangular, hexagonal, or (in few cases) random, the important issue is the neighbor-property of neurons which defines a distance function on the grid: two neurons have grid-distance $k$ if the shortest connection between the neurons has $k$ connection lines. This means each neuron has distance 0 to itself and distance 1 to all its neighbors.

Note that we have to distinguish between two different distance functions when dealing with a SOM.

1. There is the distance in the input space we need to determine the closest center for an input – usually the Euclidian distance.
CLASSIFYING NETWORKS

2. The grid distance defined by the grid topology on the finite set of neurons in the output layer.
In both cases also alternative distance functions are possible.
The training is an extension of the winner take all strategy: not only the winner changes its weights via the LVQ rule but also the surrounding neurons up to a given radius (upper bound) of the distance, where the learning rate $\eta$ is reduced for increasing grid-distance.

$$c_k = c_k + \eta_d^*(c_k - x) \quad \eta_0 > \eta_1 > \ldots > \eta_{\text{radius}} = 0$$
if the neuron $c_k$ has distance $d < \text{radius}$ from the winner.

The original Kohonen paper suggests a two-phase-learning starting with a larger radius (ordering phase) and then reducing the radius (tuning phase).
Note that there is no bound on the dimensionality of the input space, only the dimensionality of the grid is bound to 1 or 2.
In order to show the learning effect we construct a simple example with 2-dimensional input and a 4x7 grid.

```matlab
P = rands(2,1000); % input set
Q = P(:,[11:38]); % 28 inputs for the grid
net = newsom([-1 1 ;-1 1],[4 7]); % construct the SOM
    % by default hex-grid
net.IW{1}=Q'; % initialize weights with Q
plotsompos(net,P); % plot the positions of the grid
```

Figure 5-10: grid in the input space
These MATLAB commands generate a 2-dimensional random input set with all coordinates between -1 and +1, select a 28 (=4x7) subset, construct a SOM with the 2-dimensional input space with coordinates between -1 and +1 and a 4x7 grid, initialize the weights with the 28 selected points, and finally shows the position of the 28 centers with their grid connections in the input space with the input set P in the background. The random choice of the points in P and Q has as consequence that the grid structure is strangely folded into the input space.
Since SOM training does not only move the winner but also its grid-neighbors regardless of their distance to the actual input, the training has two effects:
1. The winner is moved towards the input
2. The grid structure is somewhat unfolded within the input space.

After a successful training we expect a faithful picture of the grid evenly spreading between the training inputs.

Let us look at a more interesting example with 4 obvious clusters to explain the excellent plotting features of SOM:

```matlab
load simplecluster_dataset;
S = simplecluster_dataset;
net = newsom(S,[9 11]);
net = train(net,S);
plotsompos(net,S);
```

Figure 5-11: trained SOM (10,000 epochs)

```matlab
plotsomtop(net); % topology
plotsomnc(net); % neighbor connection
```

give a look on the grid topology on the SOM:

Figure 5-12: trained SOM (4 clusters)
Figure 5-13: topology and neighbor connection

`plotsomnd(net); % neighbor distance`
gives a look on the distances between neighbors in the input space (dark colors mean great distances, so light spots indicate clusters):

Figure 5-14: neighbor distances

`plotsomhits(net,S); % inputs per winner neuron`
give a look on number of inputs from S having a neuron as winner:

Figure 5-15: Histogram
CLASSIFYING NETWORKS

plotsomplanes(net);  % coordinates of the centers
plots for every dimension of the input a colored grid map indicating the value of the center coordinates:

Figure 5-16 : planes

SOMs show their real potential in higher input dimension, where they represent a topological embedding of the grid-structure into the high dimensional input space. Even though the pictures are not that meaningful any more they still give some insight into the distribution of the training inputs within the input space.

A SOM is not only a very useful neural network in its own right, but it can equally well serve as a front end in combination with a MLP as a network realizing separation of input and output.

5.3 Distance and scalar product
We close this section with a short remark about the coincidence of minimal distance and maximal scalar product in the winner take all strategy.

Figure 5-17 : spherical representation
In data analysis it often happens that high dimensional data points have about the same length so they can be represented as points on a sphere. If we can at least assume that all the centers $c_k$ (lines of the weight matrix) have about the same
length $L$ we have a simple numerical dependency between the distance of an input $x$ to the center and its scalar product with the center:

$$\text{dist}(x, c_k)^2 = (x - c_k)^2 = x^2 + c_k^2 - 2x \cdot c_k = x^2 + L^2 - 2x \cdot c_k$$

This formula shows, that finding the minimal distance of an input $x$ to the centers boils down to finding the maximal scalar product between $x$ and the centers. This is the reason, why many neural network tools make no distinction between the two versions of winner identification, although they may produce slightly different results in case the centers we construct have very different length.
APPENDIX

6 APPENDIX
APPENDIX

6.1 Links

6.1.1 Neural networks

- [Wikipedia page](#) on neural networks
- Introduction to neural networks ([Willamette U.](#))
- Neural Network Toolbox [Design Book](#)
- Weizmann Institute: MATLAB NN_Toolbox [Documentation](#)

6.1.2 Neural Network Commercial Software

- [Alyuda NeuroIntelligence](#), supports all stages of neural network design and application.
- [BioComp iModel(tm)](#), self-optimizing, non-linear predictive model development.
- [BrainMaker](#), a fast neural network system, now with MMX acceleration.
- [COGNOS 4Thought](#), predictive modeling tool offering Effectiveness measurement, What-if analysis, and Forecasting
- [GMDH Shell](#), creates classifiers based on GMDH-type neural networks.
- [EasyNN](#) Build a Neural Network in Five Steps
- [SwingNN](#) Learn from the past
- [KnowledgeMiner](#), 64-bit, multi-core support software for building reliable analytical classification models and model ensembles from noisy data using self-organizing knowledge mining technologies. Model export to Excel. Localized for English, Spanish, German. Free to try.
- [MemBrain](#), a powerful neural network editor and simulator for Microsoft Windows, supporting neural networks of arbitrary size and architecture (free for private and for non-commercial use).
- [NeuroSolutions](#), powerful and flexible neural network modeling software.
- [NeuroXL](#), neural networks software for classification and prediction of simple and complex data in Excel.
- [NeuralWorks Predict](#) Predict 3.0 and Professional II/PLUS.
- [SPSS Neural Connection 2](#), with Bayesian Network, Data Output Tool, model weights and more.
- [STATISTICA Neural networks](#), comprehensive and user-friendly nn application with many charting options, network architectures and training algorithms.
- [Synapse](#), a development environment for neural networks and other adaptive systems, supporting the entire development cycle from data import and preprocessing via model construction and training to evaluation and deployment; allows deployment as .NET components.
- [Tiberius](#), a data modelling and visualisation tool for Windows. (Free to university staff and students).

6.1.3 Neural Network Free Software and Shareware
APPENDIX

- JNNS Java Neural Network Simulator, Successor of SNNS by A.Zell developed in his book “Simulation neuronaler Netze”.
- JustNN The Neural Network Part of EasyNN
- BCP MLP-Simulator, U of Alberta
- NuClass7, freeware for fast development, validation, and application of neural and conventional classifiers including multilayer perceptron, functional link net, piecewise linear net, nearest neighbor classifier, self organizing map.
- Sciengy RPF, a free application for data mining with self-organizing neural networks, with a convenient user interface and ability to work with text and data files. (Windows only)
- Sharky Neural Network, free software for playing with neural networks classification.

6.1.4 Tools

- AWK is a very good file manipulation tool, where MATLAB is not so easy.
- Bruce Barnett: AWK - A Tutorial and Introduction:
- for more sophisticated manipulations: Regular expressions
- AWK is an extension of UNIX sed sufficient for many file manipulation tasks
- MATLAB - Eine Einführung (german)
- MATLAB Scriptum, Uni Münster (german)
- MATLAB eine kurze Einführung (fh Konstanz, german)
- Cleve Moler: Numerical Computing with MATLAB (introductory chapter)
- Weizmann institute: MATLAB Documentation
- MATLAB Summary and Tutorial (U of Florida)
- Data Mining with the Java SOM Toolbox (of special interest: SOM Visualizations)
- MATLAB Neural Net Toolbox, a comprehensive environment for neural network research, design, and simulation within MATLAB
6.2 Key words

A
activation function 24, 26
approximation 8, 15, 18, 29, 31, 14, 34, 35, 40, 43, 44, 45, 46, 47, 49, 69

B
backpropagation 21, 36, 38, 40, 67, 69
Batch 31
bias 23, 24, 26, 28, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 33, 34, 36, 38, 39, 47
black box 33, 20
Block 31

C
centers 43, 45, 47, 48, 49, 52, 53, 54, 56, 59, 60
class 9, 18, 51, 52, 53, 54
correct 15, 17, 18, 19, 22, 27, 29, 32, 17, 51, 68
correctness 5, 18, 19, 29, 32, 33, 54
cycle 8, 22, 24, 25, 26, 62

data
data file 6
files 10
sets 6
data files 6, 10, 12, 14, 15, 63
data sets 6, 9, 10, 14, 15, 16, 17, 18, 19, 32, 3, 4, 34, 51
delta 29, 30, 31, 3, 4, 21
dimension 6, 7, 12, 13, 14, 15, 16, 17, 18, 19, 21, 24, 7, 14, 17, 31, 38, 43, 46, 47, 55, 56, 59
distance driven 29

E
end 14, 27, 29, 31, 5, 6, 7, 8, 9, 10, 11, 18, 22, 23, 24, 33, 53, 59
Epoch 31
error 3, 15, 17, 19, 29, 30, 31, 32, 4, 6, 7, 8, 9, 10, 11, 34, 35, 36, 37, 39, 41, 42, 47, 49, 70
mean square 46
errors 6
excitatory 22, 28

F
feed forward 27, 22, 24, 26, 30, 31, 36, 37, 41
feedback 26, 27, 21, 22, 24, 26, 27, 40

g
gauss function 14
generalization 5, 18, 19, 32
gradient descent 20, 34, 35, 36, 39, 40, 41, 42
greatest similarity 48, 52, 53, 54

H
Hebb 29, 30, 31, 68

I
inhibitory 22, 28
input oriented 29, 51

K
Kohonen Maps 54

L
layers 25, 26, 13, 15, 16, 17, 20, 21, 24, 30, 31, 32, 33, 35, 37, 38, 46, 47, 51, 53, 54, 55, 56
learn 26
learning 18, 26, 29, 32, 4, 5, 7, 8, 9, 10, 11, 12, 35, 36, 38, 39, 40, 41, 49, 50, 52, 53, 54, 56, 67, 68, 69
levels 6, 15, 18, 32, 21, 22, 24, 25, 26, 35, 36, 38, 54
local minimum 35, 41, 42
APPENDIX

6.3 Glossary
6.4 Bibliography

APPENDIX

APPENDIX

APPENDIX

APPENDIX